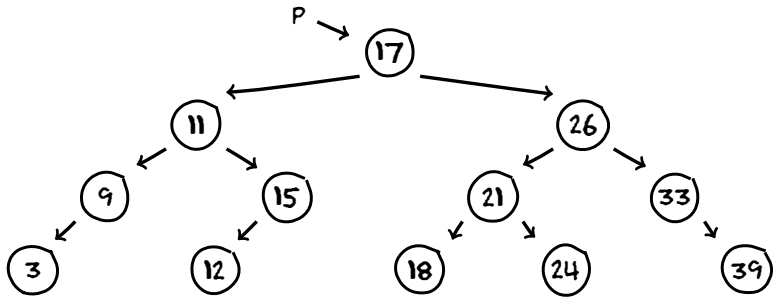
Estruturas de Dados

aula 16: Reorganização de árvores

1 Introdução

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

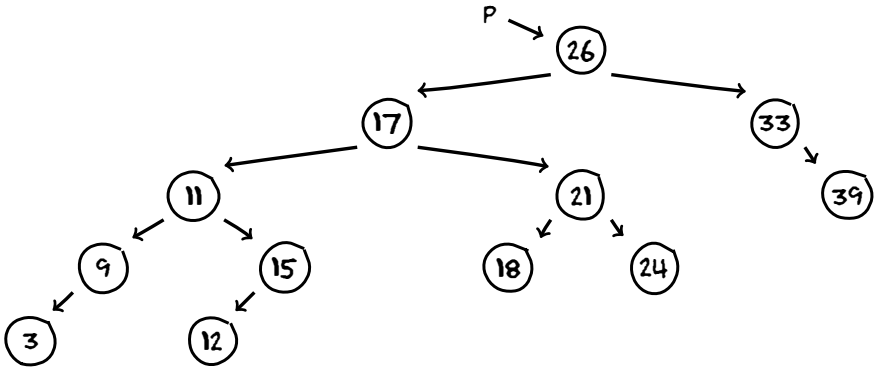


E imagine que nós queremos remover o 17 da árvore.

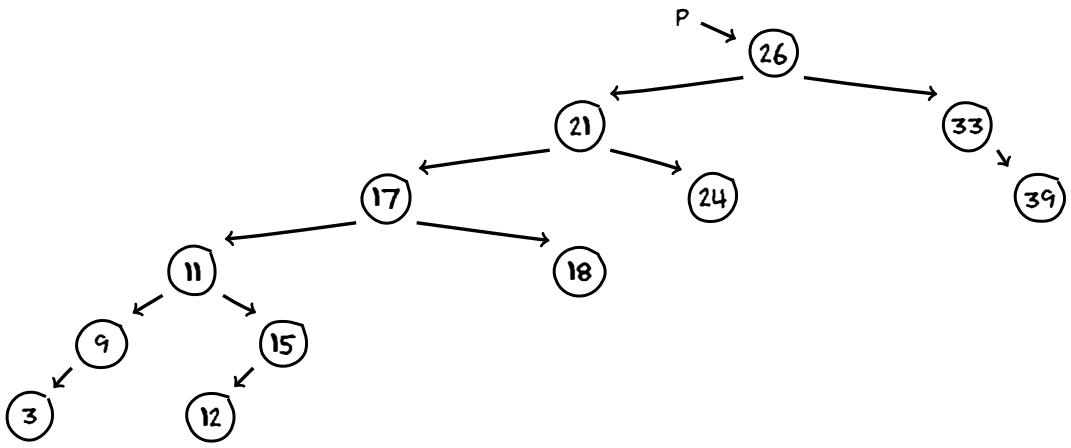
Mas, nós não queremos colocar o menor elemento da subárvore direita no lugar dele.

Como é que a gente faz isso?

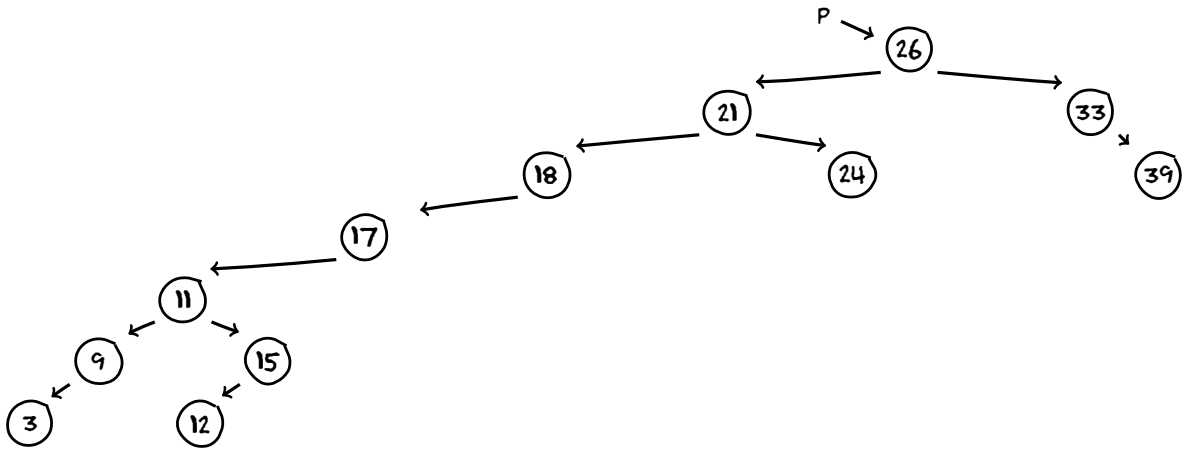
Bom, uma ideia consiste em fazer o 26 tomar o lugar do 17 na raiz da árvore



Daí, nós podemos fazer o 21 tomar o lugar do 17

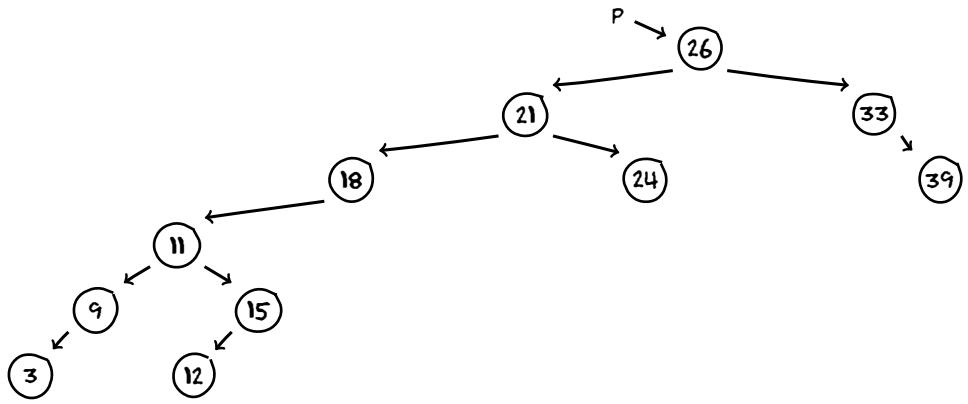


E depois, nós colocamos o 18 no lugar do 17



Agora, o 17 não tem mais um filho direito.

Logo, ele pode ser removido da árvore colocando o filho esquerdo no seu lugar



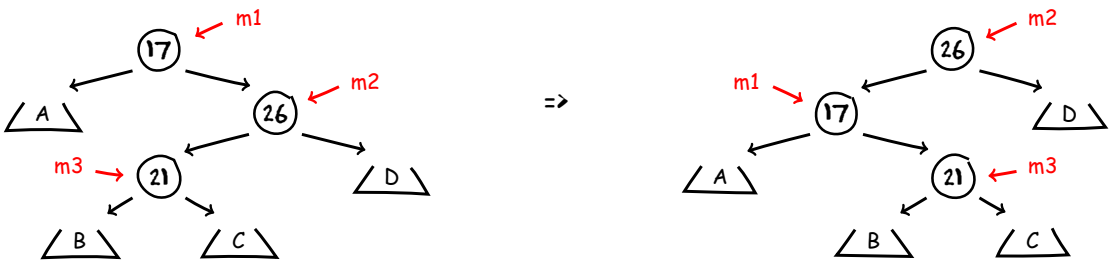
Sim! nós descobrimos outro algoritmo para fazer a remoção na árvore binária de busca.

E a sua ideia é bem simples

1. Fazer o filho direito tomar o lugar do elemento, até que não exista mais um filho direito.
2. E daí, remover o elemento da árvore

A operação importante acontece no passo 1: *o filho toma o lugar do pai.*

E nós observamos que ela pode ser implementada com a ajuda de 3 mãozinhas



— (*e apenas 2 instruções*)

Agora, nós podemos colocar essa operação dentro de uma função auxiliar

```
func SFD (q)                                                    // Sobe Filho Direito
     Se (q == Nulo   OU   q->dir == Nulo)
          Retorna ( q )
     Seão
          m1 = q;   m2 = m1->dir;   m3 = m2->esq                // coloca as mãozinhas
          m2->esq = m1;   m1->dir = m3                          // e rearranja
          Retorna (m2)
```

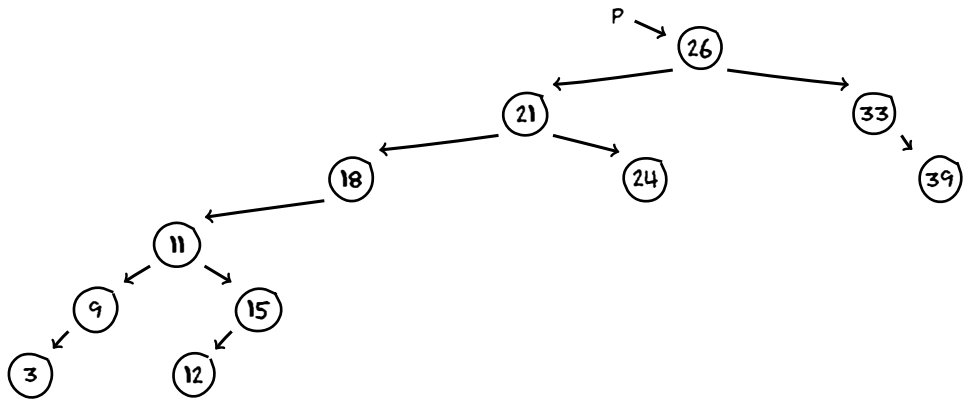
E a função de remoção fica assim

```
func remoção-Rec (q,k)
     Se (q == Nulo)  Retorna ( q )                              // já removi
     Sse (q->num > k)   q->esq = remoção-Rec ( q->esq,k )       // descendo ...
     Sse (q->num < k)   q->dir = remoção-Rec ( q->dir,k )       // descendo ...
     Sse (q->dir  ==  Nulo)
          fe = q->esq;   free(q)                                // remoção do elem
          Retorna (fe)
     Senão
          q = SFD(q)                                            // sobe o filho
          q->esq = remoção-Rec ( q->esq,k )                     // descendo ...
```

2 Reorganização de árvores

O truque que nós acabamos de ver é legal.

Mas ele deixa a árvore completamente desbalanceada



Só que isso é uma coisa fácil de resolver.

Quer dizer, olhando para a árvore nós vemos que

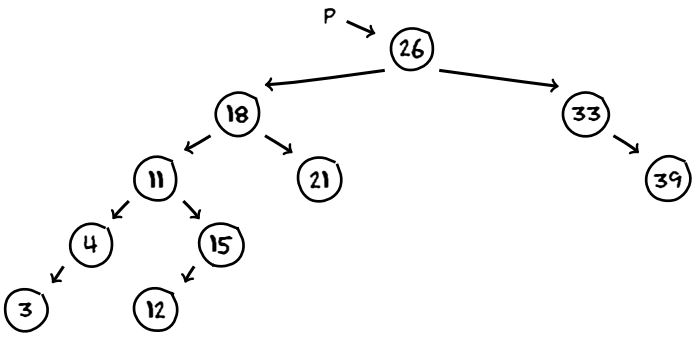
- existem 5 elementos maiores que o 18
- existem 4 elementos menores que o 18

Então, parece uma boa ideia colocar o 18 na raiz da árvore.

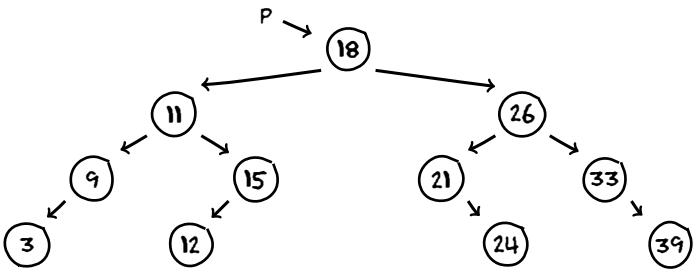
Mas, como é que a gente faz isso?

Bom, basta fazer o 18 tomar o lugar do seu pai sucessivamente, até ele chegar na raiz da árovre.

Primeiro o 18 toma o lugar do 21

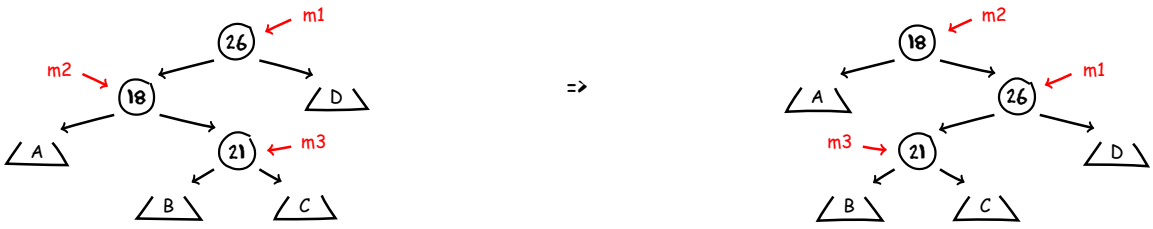


E depois o 18 toma o lugar do 26



— (*você viu o que aconteceu, no final das contas?*)

Para implementar essa ideia, basta escrever a função que faz o filho esquerdo tomar o lugar do pai




```
func SFE (q)                                        // Sobe Filho Esquerdo
     Se (q == Nulo   OU   q->esq == Nulo)
          Retorna ( q )

     Seão
          m1 = q;   m2 = m1->esq;   m3 = m2->dir    // coloca as mãozinhas
          m2->dir = m1;   m1->esq = m3              // e rearranja
          Retorna (m2)
```

E agora, nós estamos prontos para escrever a função que encontra um elemento e faz ele subir até a raiz

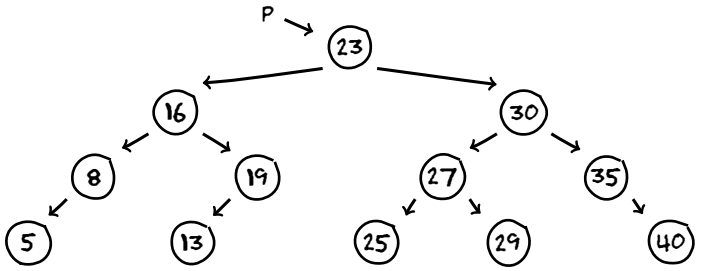
```
func SAR (q,k)                                          // Sobe Até a Raiz
     Se (q == Nulo)  Retorna ( Nulo )                   // Não está lá ...
     Sse (q->num > k)
          aux = SAR (q->esq,k)
          Se (aux != Nulo)
               q->esq = aux;   q = SFE (q)
               Retorna (q)
     Sse (q->num < k)
          aux = SAR (q->dir,k)
          Se (aux != Nulo)
               q->dir = aux;   q = SFD (q)
               Retorna (q)
     Senão  Retorna (q)
```

A seguir nós vamos ver mais exemplos.

Exemplos

a. Partição da árvore

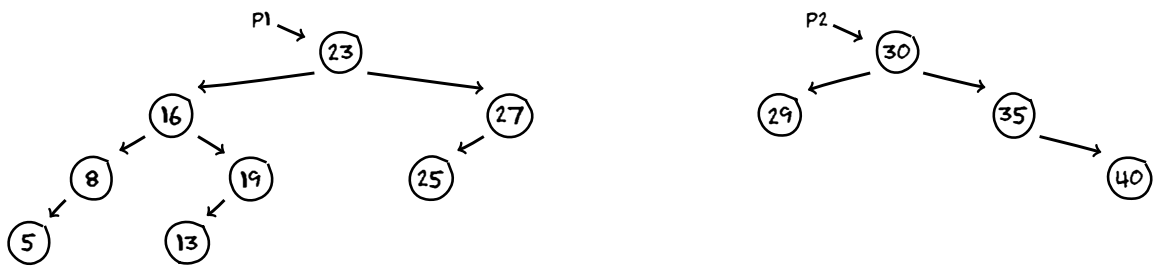
Imagine que o ponteiro p aponta para uma árvore binária de busca



A tarefa é

→ *particionar a árvore em duas, onde a primeira contém os elementos* $\leq$ k*,*
e a segundo conté os elementos $>$ k

No exemplo acima, para $k = 28$, isso nos daria



Legal.

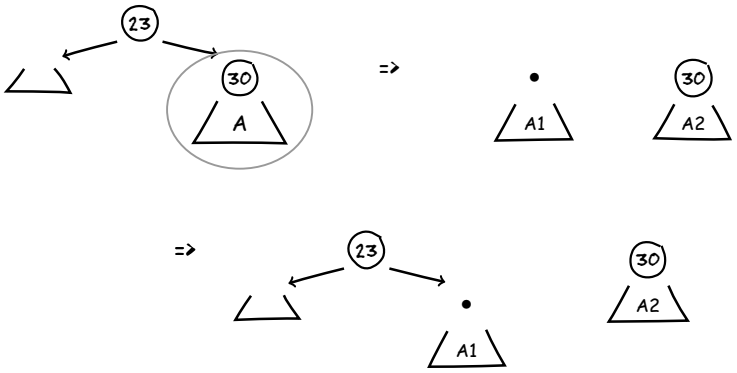
Mas, como é que a gente faz isso?

Bom, vamos começar olhando para a raiz.

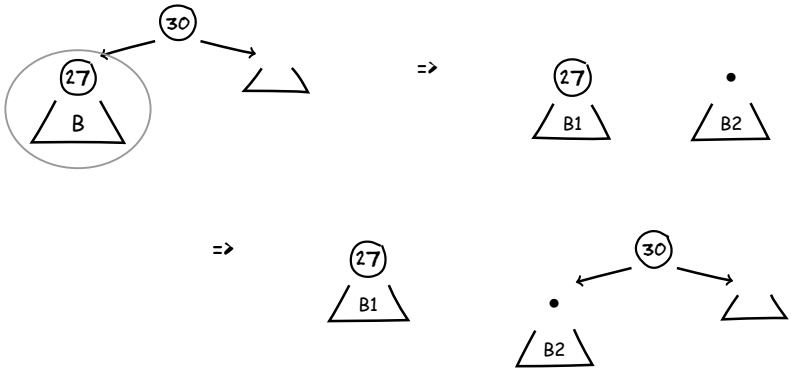
Nesse caso, como a raiz é igual a 23, ela vai ficar na primeira parte da árvore — (*juntamente com a sua subárvore esquerda*)

Daí, a gente particiona a subárvore direita.

A primeira subárvore dessa partição vai ser o filho direito do 23 — (*e a recursão vai acontecer na segunda parte*)



E daí, a gente nota que a partição da subárvore do 30 é análoga



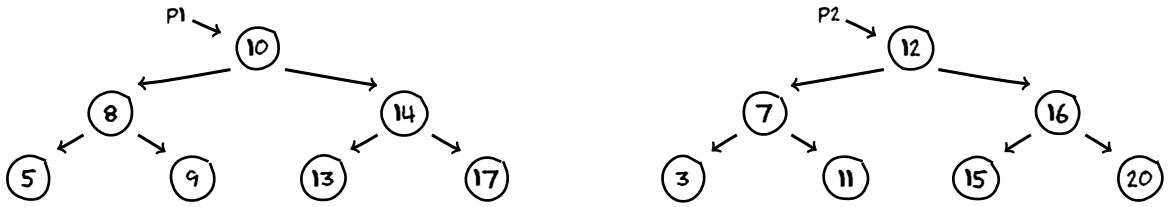
Agora, já não é difícil escrever a função

```
func partição-Rec (q,k)
     Se (q == Nulo)  Retorna ( Nulo, Nulo )
     Sse (q->num ≤ k)
          p1 = q
          a1,a2 = partição-Rec (q->dir,k)
          p1->dir = a1;   p2 = a2
     Senão
          p2 = q
          b1,b2 = partição-Rec (q->esq,k)
          p2->esq = b2;   p1 = b1
     Retorna (p1,p2)
```

◇

b. Intercalação de árvores

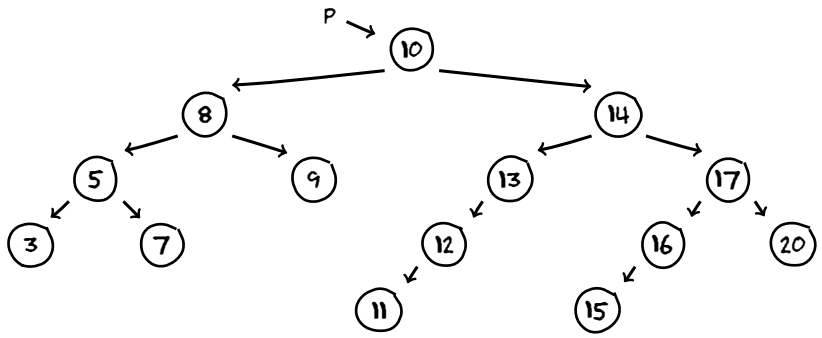
Imagine que os ponteiros p1, p2 apontam para duas árvores binárias de busca



A tarefa é

→ *produzir uma única árvore binária de busca com os elementos de* p1 *e* p2

No exemplo acima, isso nos daria



Legal.

Mas, como é que a gente faz isso?

Bom, faz sentido escolher a raiz de uma das duas árvores para ser a raiz da árvore final.

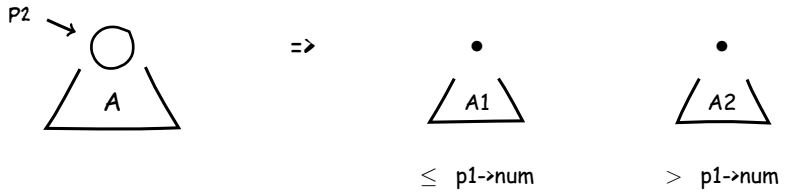
Nesse raciocínio, nós vamos escolher a raiz de p1.

Daí, é bem fácil ver que

- os elementos da subárvore esquerda de p1 vão ficar do lado esquerdo
- os elementos da subárvore direita de p1 vão ficar do lado direito

Mas, o que é que a gente faz com p2*?*

Bom, faz sentido particionar p2 utilizando a raiz de p1



Daí,

- a primeira parte vai ficar do lado esquerdo
- a segunda parte vai ficar do lado direito

E agora basta

- intercalar as duas subárvores que vão ficar do lado esquerdo
- intercalar as duas subárvores que vão ficar do lado direito

A coisa fica assim

```
func intercalação-Rec (p1,p2)

     Se (p1 == Nulo)   Retorna ( p2 )

     Se (p2 == Nulo)   Retorna ( p1 )

     Senão
          raiz = p1
          a1 = p1->esq;   a2 = p1->dir
          b1,b2 = partição-Rec (p2,p1->num)

          raiz->esq = intercalação-Rec (a1,b1)
          raiz->dir = intercalação-Rec (a2,b2)

          Retorna (raiz)
```

◇

c. Inserindo a árvore na árvore

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

Quer dizer, a gente também pode fazer uma função que insere a árvore p2 dentro da árvore p1.

A ideia é a seguinte.

Imagine que X é a raiz da árvore p1, e Y é a raiz da árvore p2

E imagine que X > Y.

Então, nós podemos decompor a árvore p2 assim

Daí, nós inserimos a primeira parte no lado esquerdo de p1.

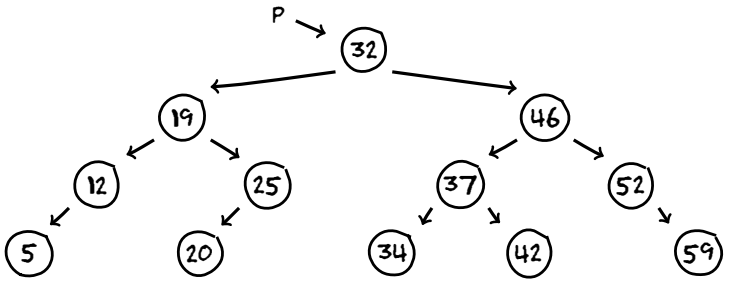
E fazemos uma chamada recursiva com a segunda parte e a árvore resultante dessa operação.

A coisa fica assim

```
func IAA-Rec (q1,q2)                                    // Inserção da Árvore na Árvore
     Se (q1 == Nulo)   Retorna ( q2 )
     Se (q2 == Nulo)   Retorna ( q1 )
     Sse (q1->num > q2->num)
          b1 = q2;   b2 = q2->dir
          q2->dir = Nulo
          q1->esq:= IAA-Rec (q1->esq,b1)
          resp:= IAA-Rec (q1,b2)
          Retorna (resp)
     Senão
          // análogo ao caso anterior ...
```

d. Separando pares e ímpares

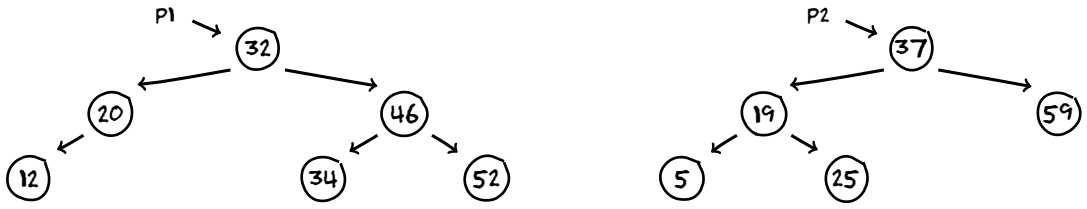
Imagine que o ponteiro p aponta para uma árvore binária de busca



A tarefa é

→ *separar os elementos pares e ímpares da árvore em duas árvores* p1 *e* p2

No exemplo acima, isso nos daria



Certo.

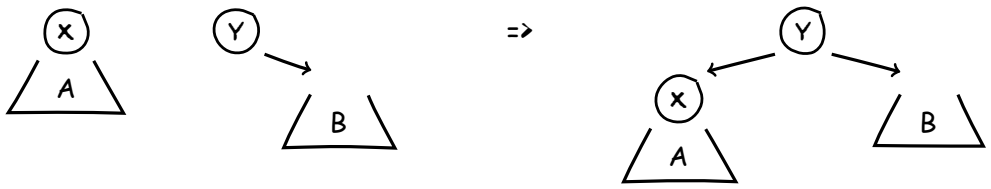
Mas, como é que a gente faz isso?

Bom, nesse exemplo a raiz da árvore é um número par

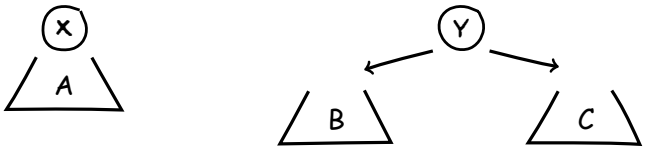
Então, depois de separar os pares e ímpares nas subárvores esquerda e direita, nós podemos montar a árvore para com a raiz da árvore e as subárvores pares de cada lado.

Mas, como a gente monta a árvore ímpar?

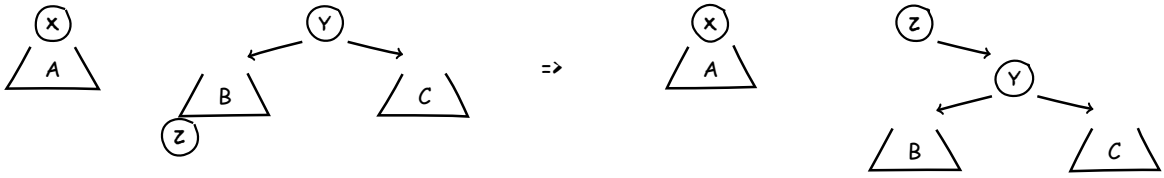
Bom, o caso em que uma das subárvores ímpares está vazia é fácil — (*basta retornar a outra*)
E o caso em que a subárvore ímpar da direita não tem o filho esquerdo também é fácil



Então agora, a gente considera o caso em que a subárvore ímpar da direita tem o filho esquerdo



Bom, nesse caso a gente pode remover o menor elemento do lado esquerdo (B), e colocá-lo na posição da raiz

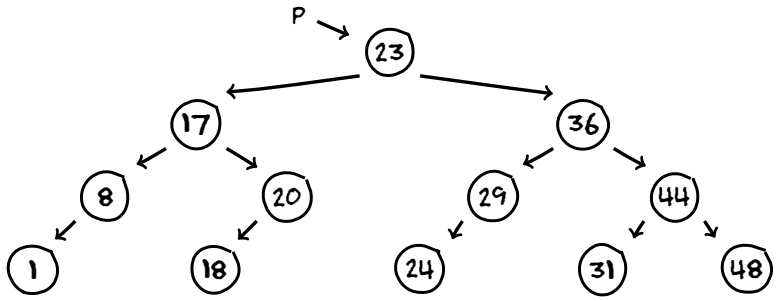


E agora nós já sabemos o que fazer.
A coisa fica assim

```
func SPI-Rec (q)                                    // Separando Pares e Ímpares
      Se (q == Nulo)   Retorna ( Nulo, Nulo )
      a1,b1 = SPI-Rec (q->esq)
      a2,b2 = SPI-Rec (q->dir)
      Se (q->num é par)
            q->esq,dir = a1,a2
            Se (b1 == Nulo)   Retorna (q,b2)
            Sse (b2 == Nulo)   Retorna (q,b1)
            Sse (b2->esq == Nulo)
                  b2->esq:= b1;   Retorna (q,b2)
            Senão
                  z,b2 = removeMenor (b2)
                  z->esq,dir = b1,b2;   Retorna (q,z)
      Senão
      // análogo ao caso anterior ...
```

e. Cortando no nível k

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

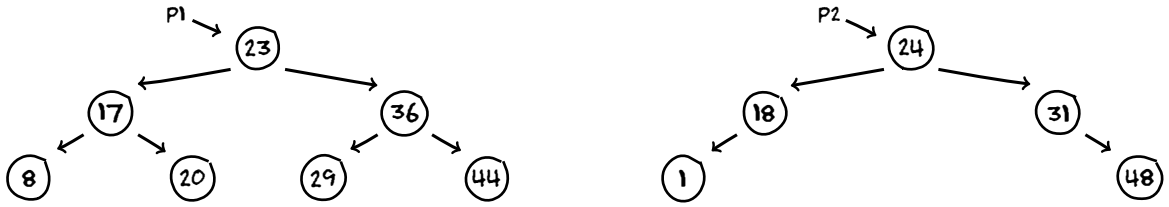


A tarefa é

→ *remover todos os elementos que estão abaixo do nível* k

colocando os elementos removidos em outra árvore

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

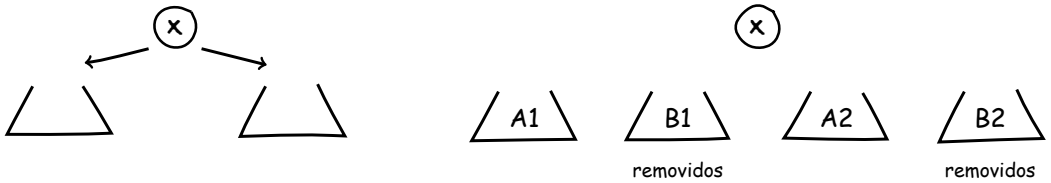


Certo.

Mas como é que a gente faz isso?

Bom, a gente faz isso raciocinando recursivamente.

Quer dizer, imagine que os elementos abaixo do nível k nas subárvores esquerda e direita já foram removidos — (*e já foram colocados em subárvores separadas*)



Então, a solução do problema fica assim

E a gente usa o truque do exemplo anterior para juntar as árvores B_1 e B_2

Além disso, nós também utilizamos um parâmetro para indicar o nível em que a gente está na recursão.

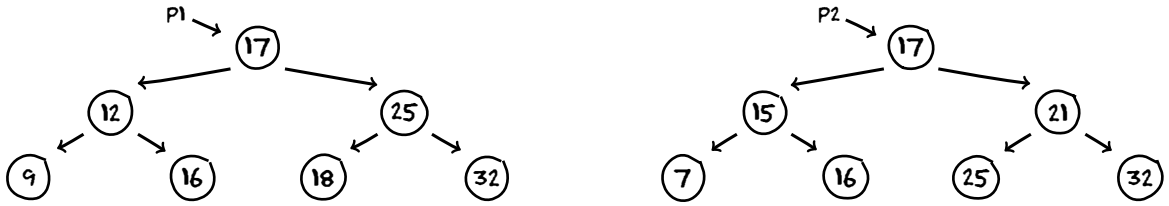
A coisa fica assim

```
func CNNk-Rec (q,k,nivel)                                    // Corte No Nível K
     Se (q == Nulo)   Retorna ( Nulo, Nulo )
     Se (nivel > k)   Retorna ( Nulo, q )
     Senão
          a1,b1 = CNNK-Rec (q->esq,k,nivel+1)
          a2,b3 = CNNK-Rec (q->dir,k,nivel+1)
          q->esq,dir = a1,a2
          r = juntaArv (b1,b2)
          Retorna (q,r)
```

◇

f. A operação de interseção

Imagine que os ponteiros p1 p2 apontam para duas árvores binárias de busca

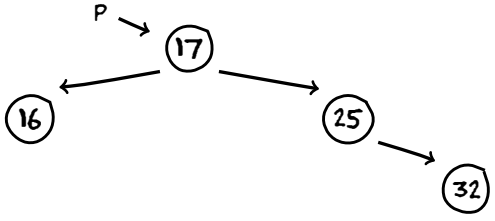


A tarefa é

→ *construir uma árvore que contém uma cópia dos elementos que estão em* p1 *e* p2

— *(sem modificar as árvores* p1 *e* p2*)*

No exemplo acima, isso nos daria



Certo.

Mas, como é que a gente faz isso?

Bom, o caso mais fácil é aquele onde as raízes de p1 e p2 são iguais.

Porque daí, nós criamos uma cópia da raiz.

E fazemos as interseções dos dois lados para completar a solução.

Agora, imagine que a raiz de p1 é maior do que a raiz de p2.

Nesse caso, nós demontamos a árvore apontada por p2 da seguinte maneira

Daí, nós fazemos a interseção da primeira parte de p2 com o lado esquerdo de p1.

E fazemos a interseção da segunda parte de p2 com a árvore p1 inteira — (*porque podem haver elementos de* B_2 *que estão em* A_1)

E daí, nós juntamos as duas árvores resultantes.

A coisa fica assim

```
func interseção-Rec (q1,q2)

     Se (q1 == Nulo  OU  q2 == Nulo)   Retorna ( Nulo )

     Sse (q1->num == q2->num)
          r = Novo (RegInt)
          r->num = q1->num
          r->esq = interseção-Rec (q1->esq,q2->esq)
          r->dir = interseção-Rec (q1->dir,q2->dir)
          Retorna (r)

     Sse (q1->num > q2->num)
          b1 = q2;   b2 = q2->dir
          q2->dir = Nulo
          a1 = interseção-Rec (q1->esq,b1)
          a2 = interseção-Rec (q1,b2)
          r = juntaArv (a1,a2)
          Retorna (r)

     Senão
          // análogo ao caso anterior ...
```

◇