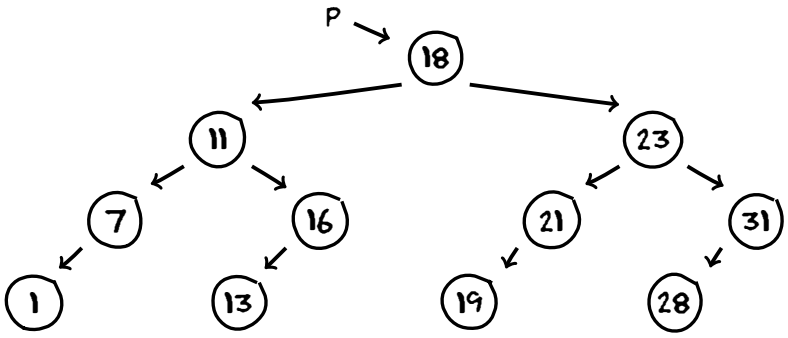
Estruturas de Dados

aula 15: Algoritmos iterativos para árvores

1 Introdução

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



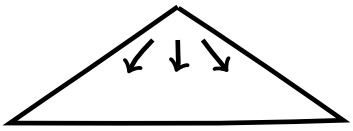
Imagine que a tarefa é

→ *imprimir todos os elementos da árvore*

E imagine que nós queremos realizar a tarefa com um algoritmo iterativo — (*com um dedo que percorre a árvore*)

Como é que a gente faz isso?

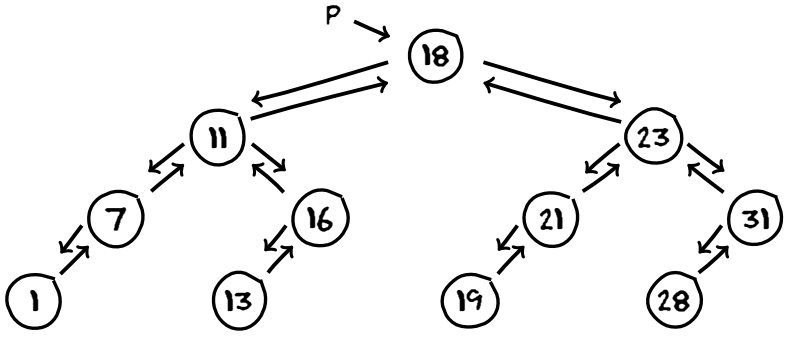
Bom, o primeiro problema é que a árvore só tem ponteiros para baixo



Logo, depois que o dedo desce, ele não tem como subir.

Só que, isso é fácil de resolver.

Quer dizer, basta fazer com que cada elemento aponte para o seu pai — (*e a raiz apoonte para* `Nulo`)



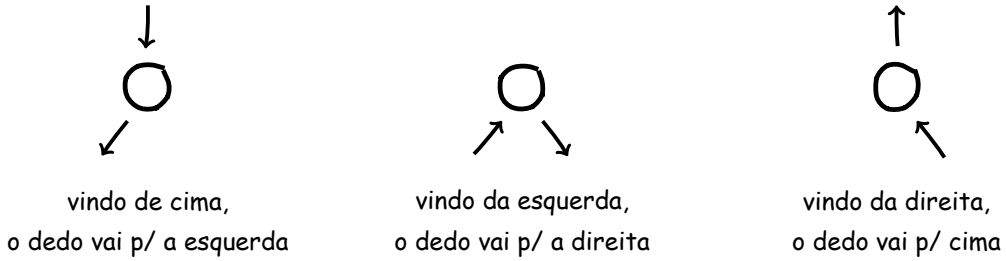
Mas isso ainda não resolve todos os problemas.

Quer dizer, na lista encadeada o dedo só pode ir pra frente e pra trás.

Mas na árvore existem várias maneiras de descer.

E na hora de subir, é preciso saber se a gente vai continuar subindo, ou se a gente vai descer para o outro lado.

Na prática, existem 3 casos



Então, se a gente sabe daonde o dedo está vindo, a gente também sabe pra onde o dedo tem que ir.

E para implementar essa ideia, a gente utiliza dois ponteiros

- `q`: o dedo que percorre a árvore
- `qant`: o lugar onde o dedo estava no passo anterior

No início, nós temos

`q = p` e `qant = Nulo`

E o processo continua enquanto o `q` é diferente de `Nulo`.

A coisa fica assim

```
func percorre-Arv (p)

     q = p;   qant = Nulo
     Enquanto ( q != Nulo )

          Se (qant == q->pai)                    // Caso 1: vindo de cima
               qant = q
               Se (q->esq != Nulo)    q = q->esq
               Sse (q->dir != Nulo)   q = q->dir
               Senão                  q = q->pai

          Se (qant == q->esq)                    // Caso 2: vindo da esquerda
               qant = q
               Se (q->dir != Nulo)   q = q->dir
               Senão                  q = q->pai

          Senão                                  // Caso 3: vindo da direita
               qant = q;   q  = q->pai
```

Legal.

Mas essa função não faz nada — (*ela só percorre a árvore*)

Só que, é bem fácil modificá-la para que ela faça alguma coisa.

Quer dizer, a gente quer imprimir os elementos da árvore.

E quer fazer isso em ordem crescente.

Então, cada elemento deve ser impresso depois que a sua árvore esquerda já foi percorrida.

Isso significa que nós vamos imprimir o elemento quando nós estivermos subindo pelo filho esquerdo — (*i.e., no caso 2*)

Ou então ao chegar no elemento, quando ele não tiver um filho esquerdo — (*i.e., no caso 1*)

A coisa fica assim

```
func imprime-Arv (p)

     q = p;   qant = Nulo
     Enquanto ( q != Nulo )

          Se (qant == q->pai)                          // Caso 1: vindo de cima
               qant = q
               Se (q->esq != Nulo)    q = q->esq
               Senão
                    Imprime (q->num)
                    Se (q->dir != Nulo)   q = q->dir
                    Senão                 q = q->pai

          Se (qant == q->esq)                          // Caso 2: vindo da esquerda
               Imprime (q->num)
               qant = q
               Se (q->dir != Nulo)   q = q->dir
               Senão                  q = q->pai

          Senão                                        // Caso 3: vindo da direita
               qant = q;   q  = q->pai
```

2 Algoritmos iterativos para árvores

Agora que nós já sabemos percorrer a árvore com o dedo, a gente pode fazer qualquer coisa.

Por exemplo, imagine que a tarefa é

→ *calcular a altura da árvore*

Bom, a ideia aqui é utilizar uma variável `Alt` que começa com o valor 1.

Daí, toda vez que a gente desce o valor de `Alt` é incrementado de 1.

E toda vez que a gente sobe o valor de `Alt` é decrementado de 1.

Ao longo do processo, a gente lembra qual foi o maior valor que a variável `Alt` assumiu.

E essa é a altura da árvore.

A coisa fica assim

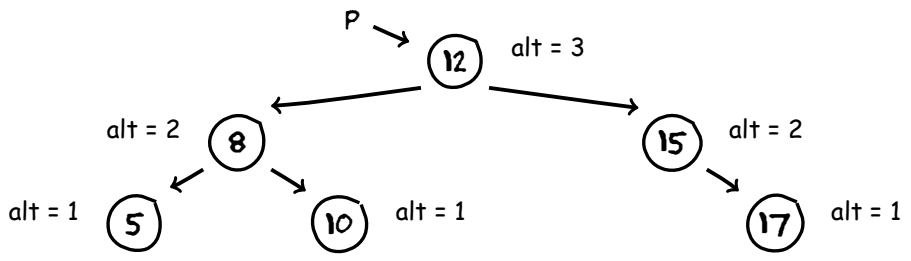
```
func altura-Arv (p)

     q = p;   qant = Nulo
     Alt = 1;   maxAlt = 0

     Enquanto ( q != Nulo )                                          // Caso 1
          Se (qant == q->pai)
               qant = q
               Se (q->esq != Nulo)    q = q->esq;   Alt++
               Sse (q->dir != Nulo)   q = q->dir;   Alt++
               Senão                  q = q->pai;   Alt--
          Se (qant == q->esq)                                        // Caso 2
               qant = q
               Se (q->dir != Nulo)    q = q->dir;   Alt++
               Senão                  q = q->pai;   Alt--
          Senão                                                      // Caso 3
               qant = q;   q  = q->pai;   Alt--
          Se (Alt > maxAlt)   maxAlt = Alt
     Retorna (maxAlt)
```

Exemplos

a. Pré-calculando a altura

Mas, não é preciso percorrer a árvore toda vez que nós quisermos consultar a sua altura.

Quer dizer, a ideia é que cada elemento pode armazenar a altura da (sub)árvore que está abaixo dele — (*no campo* `q->alt`)



Daí, para descobrir a altura da ávore, basta consultar o campo `.alt` da raiz — (*em tempo* $O(1)$)

Mas logo no início, é preciso calcular o valor do campo `.alt` de cada elemento.

Isso pode ser feito com uma pequena modificação do algoritmo do exemplo anterior.

E a boa notícia é que nós não precisamos mais das variáveis `Alt` e `maxAlt`.

Quer dizer, no momento de subir para o pai, cada elemento calcula a sua altura com base nas alturas dos seus filhos.

Para facilitar as coisas, a gente escreve uma função auxiliar para calcular a altura de um elemento

,

```
func calcAlt (q)

     alt = 0

     Se (q->esq != Nulo)    alt = (q->esq)->alt

     Se (q->dir != Nulo)    alt = Max {alt,(q->dir)->alt}

     q->alt = alt
```

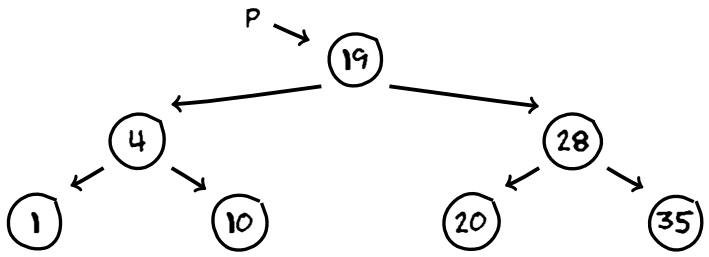
E daí, a coisa fica assim

```
func altura-Arv2 (p)

     Se (p == Nulo)

     q = p;   qant = Nulo

     Enquanto ( q != Nulo )

          Se (qant == q->pai)                                    // Caso 1
               qant = q
               Se (q->esq != Nulo)    q = q->esq
               Sse (q->dir != Nulo)   q = q->dir
               Senão                  calcAlt(q);   q = q->pai

          Se (qant == q->esq)                                    // Caso 2
               qant = q
               Se (q->dir != Nulo)    q = q->dir
               Senão                  calcAlt(q);   q = q->pai

          Senão                                                  // Caso 3
               qant = q
               calcAlt(q);   q = q->pai
```

◇

b. A operação de inserção

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



E imagine que cada elemento armazena a altura da sua subárvore.

A tarefa é

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

atualizando os campos de altura — (onde isso for necessário)

Dessa vez, a inserção será feita de maneira iterativa.

E o raciocínio fica assim

1. primeiro nós descemos até o ponto onde vai acontecer a inserção
2. depois, nós criamos o novo registro e conectamos ele à árvore
3. e daí, nós subimos outra vez até a raiz, atualizando as alturas das subárvores

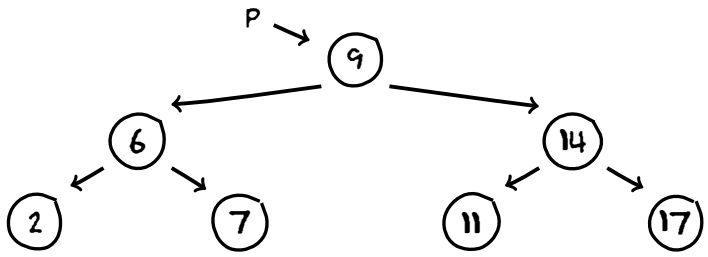
A coisa fica assim

```
func inserção-Arv (p,k)

     q = p;   qant = Nulo                        // 1a Etapa: descendo c/ 2 dedos
     Enquanto ( q != Nulo )
          qant = q
          Se (q->num > k)    q = q->esq
          Senão              q = q->dir

     r = Novo (RegInt)                           // 2a Etapa: criação do registro
     r->num,alt = k,1
     r->esq,pai,dir = Nulo,qant,Nulo
     Se (qant == Nulo)       Retorna (r)
     Sse (qant->num > k)     qant->esq = r
     Senão                   qant->dir = r

     Enquanto ( q != Nulo )                      // 3a Etapa: atualiza alturas
          calcAlt(qant)
          qant = q->pai
```

◇

c. 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, nós já sabemos que essa tarefa envolve a remoção do menor elemento de uma subárvore. Então nós começamos escrevendo a função que faz isso

```
func removeMenor-Arv (q)

     m = q;   mant = Nulo                          // descida pela esquerda
     Enquanto ( m->esq != Nulo )
          mant = m;   m = m->esq

     fd = m->dir;   fd->pai = mant                 // remoção do menor
     m->pai,dir = Nulo,Nulo

     Se (mant == Nulo)       Retorna (fd,m)
     Senão                    mant->esq = fd;   Retorna (q,m)
```

Nota: Dessa vez, nós não estamos fazendo a atualização das alturas — (*para simplificar as coisas*)

Agora, a gente imagina que também já tem a função de busca — (*que retorna um ponteiro para o elemento* `k`*, ou* `Nulo` *se ele não está lá*)

E daí, a coisa fica assim

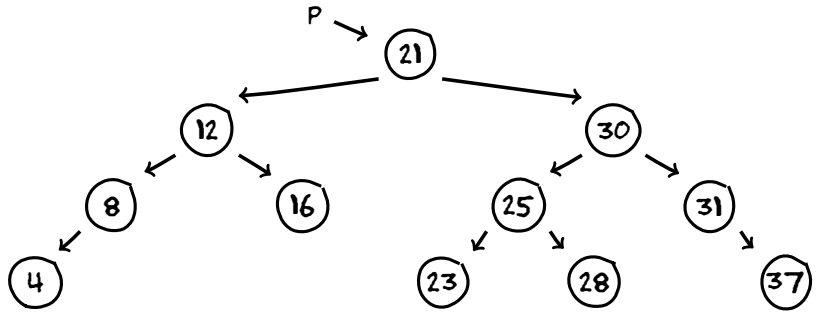
```
func remoção-Arv (p,k)

     r = busca-Arv (p,k)                          // encontra o elemento
     Se (r == Nulo)   Retorna (p,Nulo)

     Senão
          Se (r->dir == Nulo)                     // filho esq. no lugar do elemento
               fe = r->esq;   fe->pai = r->pai
               r->esq,pai = Nulo,Nulo
               Se (fe->pai == Nulo)          Retorna (fe,r)
               Sse ((fe->pai)->num > k)      (fe->pai)->esq = fe
               Senão                         (fe->pai)->dir = fe
```

```
        Senão                              // menor à dir. no lugar do elemento
              fe = r->esq;   pr = r->pai
              fd,m = removeMenor-Arv ( r->dir )
              m->esq,pai,dir = fe,pr,fd
              Se ( pr == Nulo )              Retorna ( m, r )
              Sse ( pr->num > k )            pr->esq = m
              Senão                          pr->dir = m

    Retorna ( p, r )
```

◇

d. Imprimindo o k-ésimo menor

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



A tarefa é

→ *imprimir o* k*-ésimo menor elemento da árvore*

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

```
=>  25
```

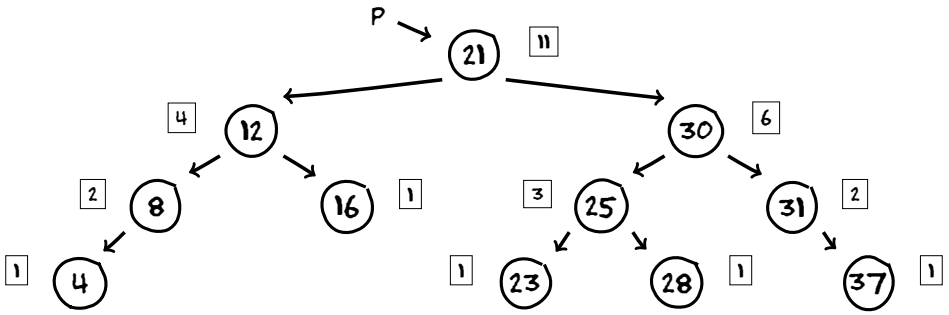
Bom, existe uma maneira simples de resolver essa tarefa.

Quer dizer, a gente pode percorrer a árvore contando os elementos por onde a gente passa.

Daí, quando o contador chega em k, a gente imprime o elemento.

Mas, a gente pode ser mais esperto do que isso — (*porque isso leva tempo* $O(n)$)

Para isso, a gente imagina que cada elemento armazena o número de elementos da sua subárvore no campo `.tam`.



Daí, se a gente quer imprimir o 7º elemento, então a gente não vai para o lado esquerdo — (*porque lá só tem 4 elemntos*)

Então, a gente vai para o lado direito.

Mas quando a gente vai para o lado direito, a gente não pode esquecer quantos elementos ficaram para trás.

Nesse caso, ficaram 5 elementos para trás — (*4 na subárvore esquerda e mais a raiz*)

Daí, somando 5 com os 3 elementos da subárvore esquerda do 30, a gente obtém 8.

E assim, a gente descobre que o 7º elemento está lá.

Então agora a gente desce para a esquerda.

E nota que dessa vez não ficou nenhum elemento para trás — (*porque?*)

Daí, somando 5 com o único elemento na subárvore esquerda de 25, a gente obtém 6.

Isso significa que o 7º elemento não está lá.

Mas nós já contamos 6 elementos menores do que o elemento atual.

Então, o elemento atual é o 7º menor elemento da árvore.

E a gente pode imprimir ele

```
=>  25
```

Em resumo, a ideia é

1. descer pela árvore na direção do k-ésimo menor,
   contando quantos elementos menores do que o atual já ficaram para trás
2. e imprimir o elemento atual quando o contador chega em k

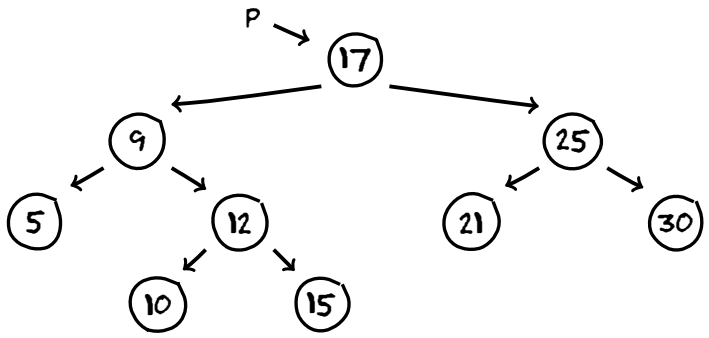
A coisa fica assim

```
func IKE-Arv (p,k)                                      // Imprime k-ésimo Elemento
     q = p;   cont = 0
     Enquanto (q != Nulo)                               // descida na árvore
          aux = 0
          Se (q->esq != Nulo)   aux = (q->esq)->tam     // filho esq. no lugar do elemento
          Se (cont + aux ≥ k)   q = q->esq
          Sse (cont + aux == k-1)
               Imprime (q->num);   Retorna
          Senão
               cont = cont + aux + 1;   q = q->dir
```

◇

e. O menor maior do que k

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



A tarefa é

→ *encontrar o menor elemento maior do que k*

No exemplo acima, para `k = 10`, isso nos daria

=> 12

Certo.

Para encontrar o menor elemento que é maior do que `k`, nós vamos fazer a descida na árvore

- o ponteiro `mmk` aponta para o candidato que nós temos no momento — (*que pode ser* `Nulo`)
- e o ponteiro `q` vai em busca de um candidato melhor

No exemplo acima, a raiz da árvore (17) já é um candidato a ser o menor elemento maior do que `k` (10) — (*e o ponteiro* `mmk` *aponta para ela*)

Daí, a gente nota que não vai encontrar nenhum candidato melhor no lado direito — (*porque?*)

Então, o ponteiro `q` vai para o lado esquerdo.

A seguir, a gente vê que o elemento apontado por `q` (9) não é maior do que `k` (10).

Logo, ele não é um candidato, e os elementos à sua esquerda também não são.

Então, o ponteiro `q` vai para a direita.

A seguir, a gente vê que o elemento apontado por `q` (12) é maior do que o `k`.

Isso nos permite atualizar o ponteiro `mmk`.

E a gente nota que não vai encontrar nenhum candidato melhor no lado direito — (*porque?*)

Então, o ponteiro `q` vai para o lado esquerdo.

A seguir, a gente vê que o elemento apontado por `q` (10) não é maior do que o `k` (10).

Logo, ele não é um candidato, e os elementos à sua esquerda também não são.

Então, o ponteiro `q` vai para a direita.

Só que, não tem ninguém do lado direito.

Logo, a coisa acaba aqui.

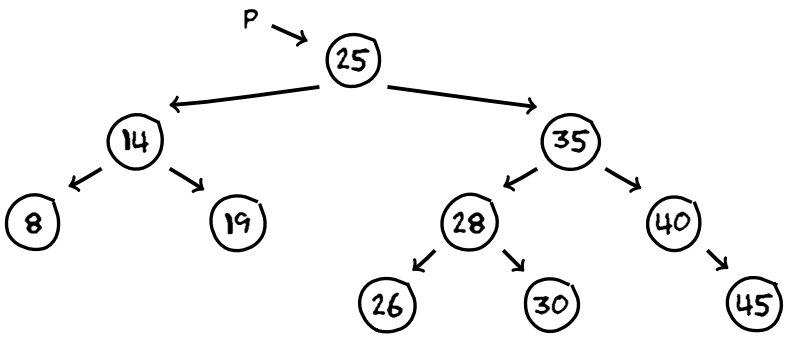
E o código da função fica assim

```
func MmK-Arv (p,k)                              // Maior menor que k

     mmk = Nulo;   q = p

     Enquanto (q != Nulo)
          Se (q->num ≥ k)   q = q->dir
          Senão
               mmk = q
               q = q->esq

     Retorna (mmk)
```

◇

f. Imprimindo uma faixa de elementos

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



A tarefa é

→ *imprimir os elementos que tem valor entre* k *e* ℓ

No exemplo acima, para k = 17 e $\ell = 37$, isso nos daria

=> 19, 25, 26, 28, 30, 35

Certo.

Nós já temos uma função que encontra o menor elemento maior que k.

Então, nós podemos colocar o dedo sobre ele, e imprimir o seu valor.

Mas, e depois?

Bom, depois a gente tem que ir para o próximo elemento.

Só que, o próximo elemento é o menor elemento que é maior do que o elemento atual.

Então, a gente pode usar a nossa função outra vez.

A coisa fica assim

```
func IFE-Arv (p,k,ℓ)                              // Maior menor que k
     q = MnK-Arv (p,k)
     Enquanto (q != Nulo)
          Imprime (q->num)
          q = MnK-Arv (p,q->num)
```

Legal.

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

Quer dizer, a gente também pode escrever uma função que encontra o próximo elemento sem começar outra vez na raiz.

Porque muitas vezes o próximo elemento vai estar ali por perto.

Existem basicamente dois casos

1. Quando o filho direito do elemento atual não é `Nulo`,

   o próximo é o menor elemento na subárvore

2. Quando o filho direito é `Nulo`, a gente precisa subir na árvore

   — (*utilizando o ponteiro pai*)

   Na subida, existem dois casos

   - Se a gente chegou no pai vindo do filho esquerdo, então o pai é o próximo elemento
   - Se a gente chegou no pai vindo do filho direito, então é preciso continuar subindo

A coisa fica assim

```
func proxEl-Arv (q)
     Se (q->dir != Nulo)                          // caso 1: descida
          px = q->dir
          Enquanto (px->esq != Nulo)
               px = px->esq
          Retorna (px)
     Senão                                        // caso 2: subida
          aux = q;   pai = q->pai
          Enquanto (pai != Nulo)
               Se (p->esq == aux)   Quebra
               Senão
                    aux = pai;   pai = pai->pai   // ha ha ha ...
          Retorna (pai)
```

E agora que nós temos essa função, nós podemos resolver o problema outra vez

```
func IFE-Arv2 (p,k,ℓ)                          // Maior menor que k

     q = MnK-Arv (p,k)

     Enquanto (q != Nulo)
          Imprime (q->num)
          q = proxEl-Arv (q)
```

◇