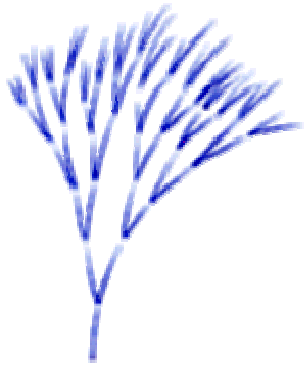
Capítulo 4 : Árvores



❒ O TAD Árvore Binária

{ No contexto das Estruturas de Dados }

Uma **Árvore Binária** é um conjunto finito de Nós, tal que:

- ou é o conjunto **vazio**,
- ou é constituída por uma **raiz** e duas Árvores Binárias distintas (chamadas **sub-árvore esquerda** e **sub-árvore direita**).

- **Operações**:

criar(T)	{ criar uma Árvore Binária vazia }
vazia(T)	{ verificar se a Árvore Binária é vazia }
construir(R, e, S)	{ construir uma AB, a partir de um elemento e duas outras ABs }
esquerda(T)	{ a sub-árvore esquerda }
direita(T)	{ a sub-árvore direita }
consultar(T)	{ consultar o elemento da raiz }

Operadores e Axiomas:

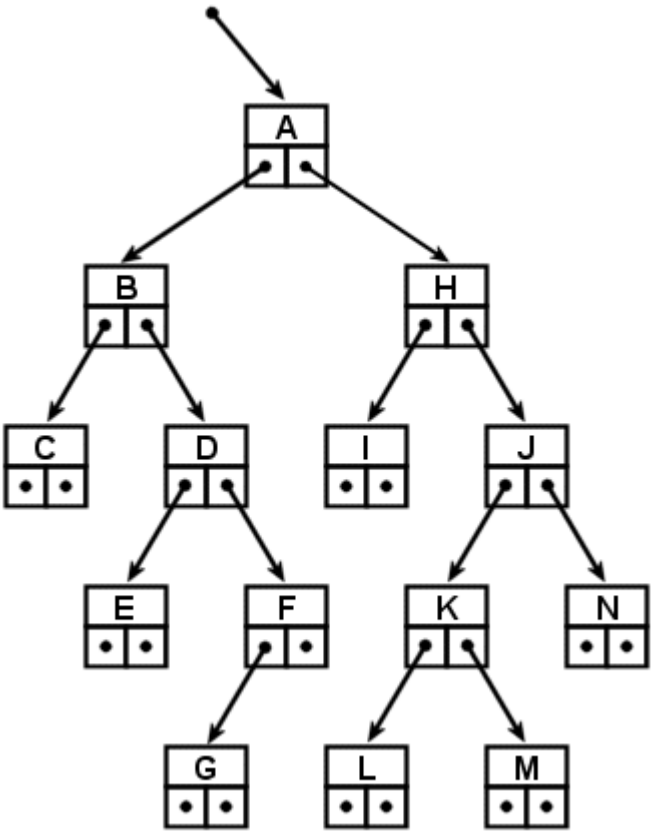
criar :	$\emptyset \rightarrow T$
vazia :	$T \rightarrow$ { verdadeiro, falso }
construir :	$T \times e \times T \rightarrow T$
direita :	$T \rightarrow T$
esquerda :	$T \rightarrow T$
consultar :	$T \rightarrow e$

vazia(criar) = verdadeiro
vazia(construir(R, e, S)) = falso
esquerda(criar) = “erro”
direita(criar) = “erro”
consultar(criar) = “erro”
esquerda(construir(R, e, S)) = R
direita(construir(R, e, S)) = S
consultar(construir(R, e, S)) = e

- **Representação**:

  O TAD Árvore Binária é facilmente representável por uma Estrutura duplamente ligada:

```
type  ArvoreBinaria  = ^no;
                  no  = record  elemento : tipo;
                                esq, dir : ArvoreBinaria
                        end;

var raiz : ArvoreBinaria;
```



Exercício: Implemente as seis operações básicas.

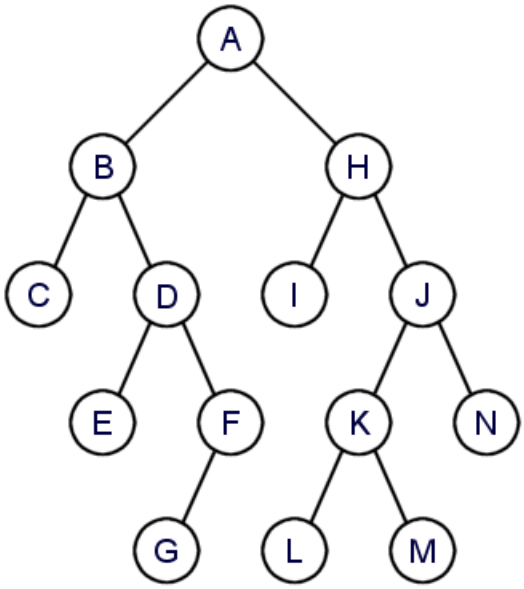
- A definição duplamente recorrente de Árvore Binária conduz, naturalmente, à construção de algoritmos duplamente recorrentes para: cópia, igualdade, travessias, pesquisas, ...
- A combinação da Recorrência Dupla com a Abstracção das Operações permite a elaboração de algoritmos simples e "compactos".

Construir uma Cópia de uma Árvore Binária:

```
R ← Cópia( T ) :
    se vazia(T)
    então  criar(R)
    senão  E ← Cópia( esquerda(T) )
           D ← Cópia( direita(T) )
           elemento ← consultar(T)
           R ← construir(E, elemento, D)
```

```
function copia (t : ArvoreBinaria) : ArvoreBinaria;
    var c, e, d : ArvoreBinaria;

    begin if vazia(t)
          then criar(c)
          else  begin e:= copia(esquerda(t));
                      d:= copia(direita(t));
                      c:= construir(e, consultar(t), d)
                end;
          copia:= c
    end; { copia }
```

- **Travessias de uma Árvore Binária**:



Pré-Ordem: 1º raiz;
2º sub-árvore esquerda;
3º sub-árvore direita.

{ A B C D E F G H I J K L M N}

Em-Ordem: 1º sub-árvore esquerda;
2º raiz;
3º sub-árvore direita.

{ C B E D G F A I H L K M J N}

Pos-Ordem: 1º sub-árvore esquerda;
2º sub-árvore direita;
3º raiz.

{ C E G F D B I L M K N J H A}

- Nas três Travessias Básicas a ordem das **"folhas"** permanece constante. Porque será?

Travessia por Níveis: **{ A B H C D I J E F K N G L M }**

- Na sua forma duplamente recorrente, a implementação das três Travessias Básicas é muito simples. Por exemplo,

```
procedure TravessiaPreOrdem (t : ArvoreBinaria);
          begin if not vazia(t)
                then begin writeln(consultar(t));
                           TravessiaPreOrdem(esquerda(t));
                           TravessiaPreOrdem(direita(t))
                     end
          end; { TravessiaPreOrdem }
```

- Contudo, as correspondentes versões iterativas são bem mais complicadas. A versão iterativa da travessia em Pré-Ordem é obtida pela utilização (quase) directa de uma Pilha, assim como a versão iterativa da travessia Por Níveis é obtida pela utilização de uma Fila auxiliar.

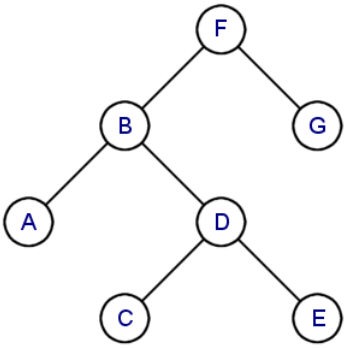
- Analisemos o caso da travessia Em-Ordem:

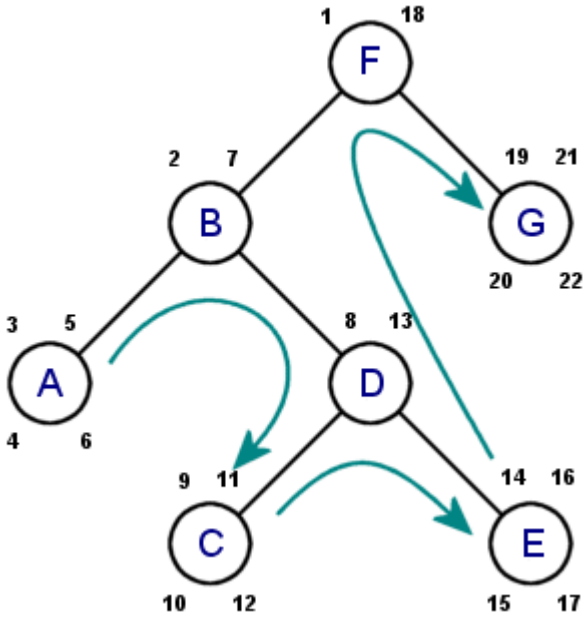
```
procedure TravessiaEmOrdem (t : ArvoreBinaria);
          begin if not vazia(t)
                then begin TravessiaEmOrdem(esquerda(t));
                           writeln(consultar(t));
                           TravessiaEmOrdem(direita(t))
                     end
          end; { TravessiaEmOrdem }
```

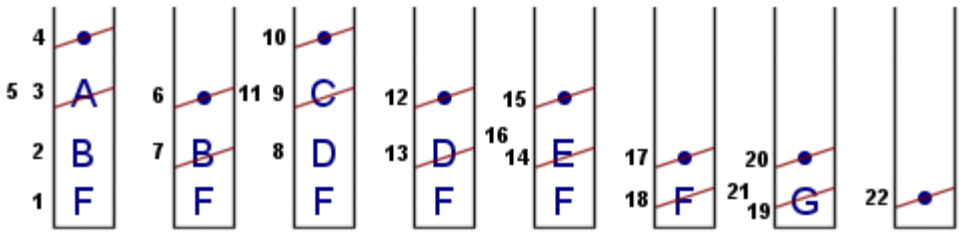
- Uma estratégia para a construção de uma versão iterativa da travessia Em-Ordem consiste na utilização de uma Pilha para guardar (os ponteiros para) as sub-árvores ainda não visitadas.

```
procedure TravessiaEmOrdemIterativa (t : ArvoreBinaria);
          var   p : ArvoreBinaria;
                S : PilhadeArvoresBinarias;
                acabou : boolean;

          begin  criar(S);
                 p:= t;
                 { começar por guardar a raiz }
                 por(p, S);
                 acabou:= false;

                 while not acabou do

                       if not vazia(p)
                       then  begin  { descer para a esquerda e guardar }
                                    p:= esquerda(p);
                                    por(p, s)
                             end

                       else  begin { subir a partir da sub-árvore vazia }
                                    tirar(S); { remover a sub-árvore vazia }

                                    { consultar a Pilha }
                                    if vazia(S)
                                    then  { toda a árvore foi visitada }
                                          acabou:= true

                                    else  begin  { visitar o topo da Pilha }
                                                 p:= topo(S);
                                                 writeln(consultar(p));
                                                 { e remover }
                                                 tirar(S);

                                                 { descer para a direita }
                                                 p:= direita(p);
                                                 { e guardar }
                                                 por(p, S)
                                          end
                             end

          end; { TravessiaEmOrdemIterativa }
```

Por exemplo,





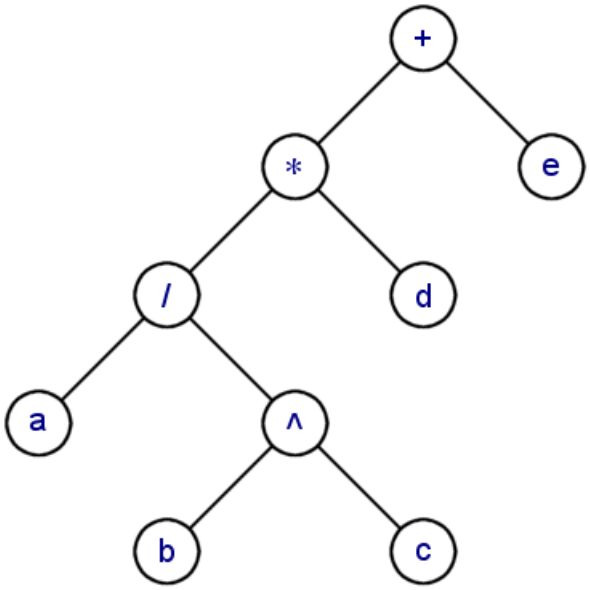


visita: A B C D E F G vazia(S)

- **Exercício**: A estratégia para a construção de uma versão iterativa para a travessia Pós-Ordem também utiliza uma Pilha auxiliar, mas é um pouco mais complicada …

Uma Aplicação das três Travessias Básicas:

- **Representação de Expressões Aritméticas.**



Travessia Pré-Ordem: + * / a ^ b c d e { Notação *prefix* }

Travessia Em-Ordem: a / b ^ c * d + e { Notação *infix* }

Travessia Pos-Ordem: a b c ^ / d * e + { Notação *postfix* }

❒ Algumas Contagens em Árvores Binárias

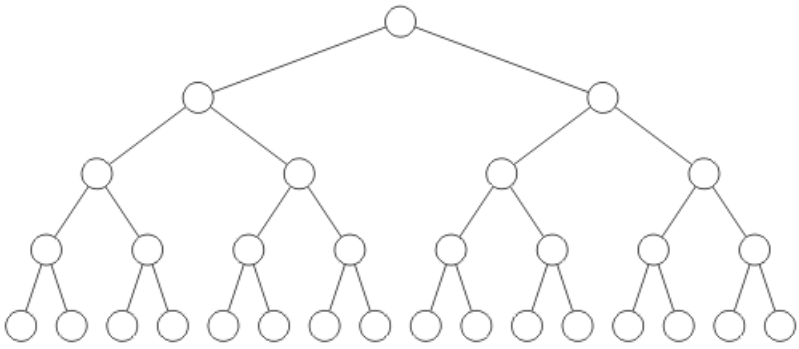
Para uma dada árvore binária, sejam:

- **n** o número total de vértices (**tamanho**)
- **e** o número de vértices externos (**folhas**)
- **i** o número de vértices internos
- **h** a altura

$$h(t) = \begin{cases} 0 & \text{se vazia(t)} \\ 1 + \max(h(\text{esquerda}(t)), h(\text{direita}(t))) \end{cases}$$

definição: Árvore binária **perfeita** (ou **total**):

$$\begin{cases} \text{uma árvore binária t de altura h = 0 é perfeita} \\ \text{uma árvore binária t de altura h > 0 é perfeita,} \\ \text{se esquerda(t) e direita(t) são perfeitas de altura h-1} \end{cases}$$



teorema: Uma árvore binária perfeita de altura h tem 2^h-1 vértices

demonstração (por Indução sobre h):

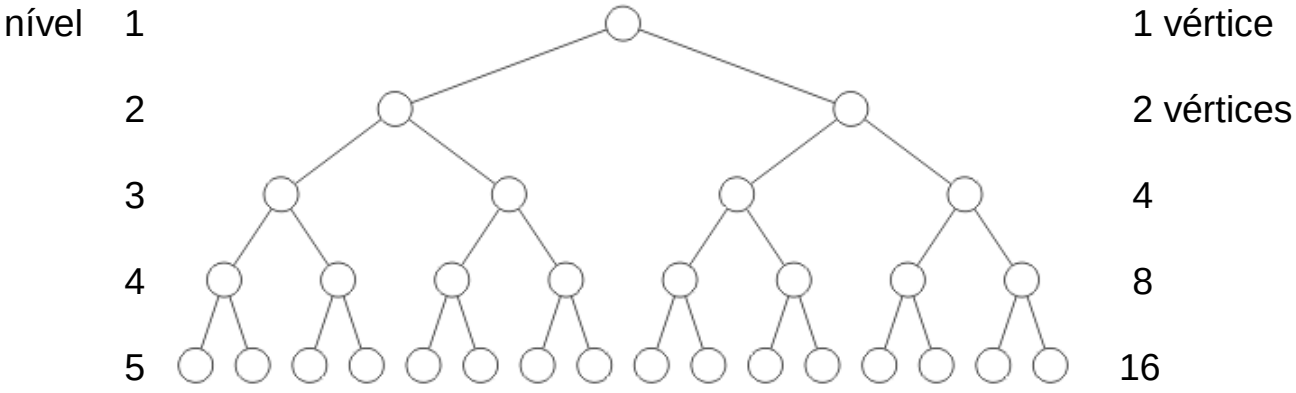
- uma árvore binária com h=0 (vazia) tem 0 vértices
- uma árvore binária perfeita de altura h+1 tem duas sub-árvores perfeitas de altura h, cada uma com 2^h-1 vértices. Portanto, $(2^h-1) + 1 + (2^h-1) = 2^{h+1} - 1$

corolário: A altura de uma árvore binária perfeita com n vértices é igual a $\log_2 (n+1)$

teorema: Uma árvore binária perfeita de altura h>0 tem 2^{h-1} folhas

demonstrar (por Indução sobre h)

corolário: Numa árvore binária perfeita, cada nível k contém 2^{k-1} vértices.



corolário: $n = 2^0 + 2^1 + \ldots + 2^{h-1} = 2^h - 1$

propriedade: Numa árvore binária perfeita, não vazia, $e = i + 1$

$$n = 2^h - 1$$
$$e = 2^{h-1}$$

$$i = n - e = 2^h - 2^{h-1} - 1 = 2^{h-1} - 1$$

propriedade: Numa árvore binária perfeita ($h>1$) mais de metade dos vértices são folhas.

$$\left.\begin{array}{l} n = 2^h - 1 \\ \\ e = 2^{h-1} \end{array}\right\} \Rightarrow \; e/n > ½$$

propriedade: Numa árvore binária perfeita, suficientemente grande, o nível médio de um vértice é cerca de $h-1$.

$$\begin{aligned} \frac{\sum_{k=1}^{h} k\ 2^{k-1}}{2^h - 1} &= \frac{2^h\ h - 2^h + 1}{2^h - 1} \\ &= \frac{2^h\ h - (2^h - 1)}{2^h - 1} \\ &= \frac{2^h\ h}{2^h - 1} - 1 \\ &= \frac{h}{1 - \frac{1}{2^h}} - 1 \\ &\approx h - 1 \end{aligned}$$

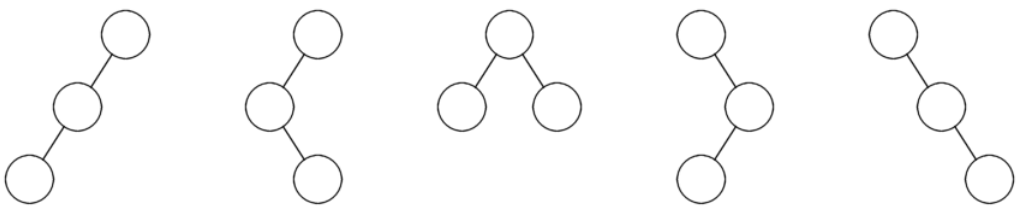
Mas nem todas as árvores binárias são perfeitas...

teorema: O **número de árvores binárias** com n vértices é igual ao Número de Catalan de ordem n,

$$C_n = \frac{(2n)!}{n!\ (n+1)!}$$

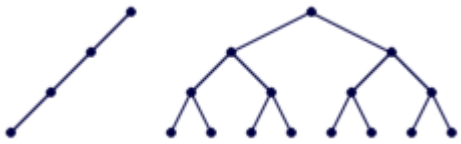
n	C_n
0	1
1	1
2	2
3	5
4	14
5	42
6	132
7	429
8	1,430
9	4,862
10	16,796
11	58,786
12	208,012
13	742,900
14	2,674,440
15	9,694,845
16	35,357,670
...	

por exemplo, para n = 3 :



teorema: Em qualquer árvore binária não vazia de altura h, com n vértices, e folhas e i vértices internos:

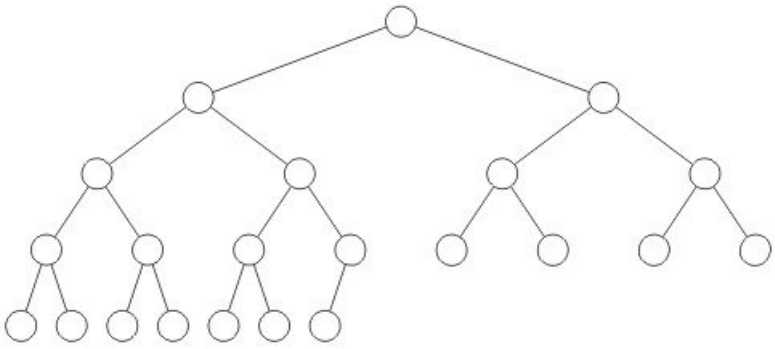
- $h \leq n \leq 2^h - 1$
- $1 \leq e \leq 2^{h-1}$
- $h-1 \leq i \leq 2^{h-1} - 1$
- $\log_2 (n+1) \leq h \leq n$



teorema: Em qualquer árvore binária não vazia, com:
n_2 o número de vértices com 2 sub-árvores não vazias,
n_1 o número de vértices com 1 sub-árvore não vazia,
n_0 o número de folhas,
então, $n_0 = n_2 + 1$.

definição: Uma árvore binária de altura h chama-se **completa** quando:

- Cada nível $k = 1 .. h-1$ contém 2^{k-1} vértices.
- Todas as folhas no nível h estão chegadas à esquerda.

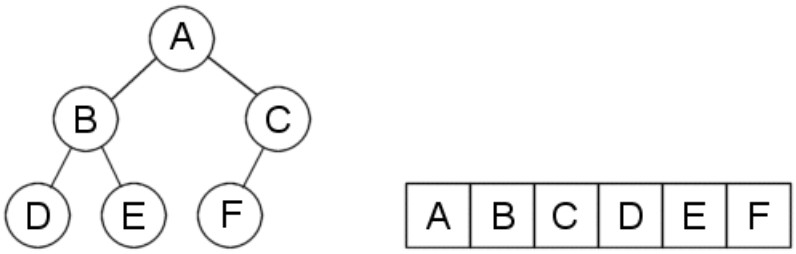


teorema: Numa árvore binária **completa** com n vértices e altura h,

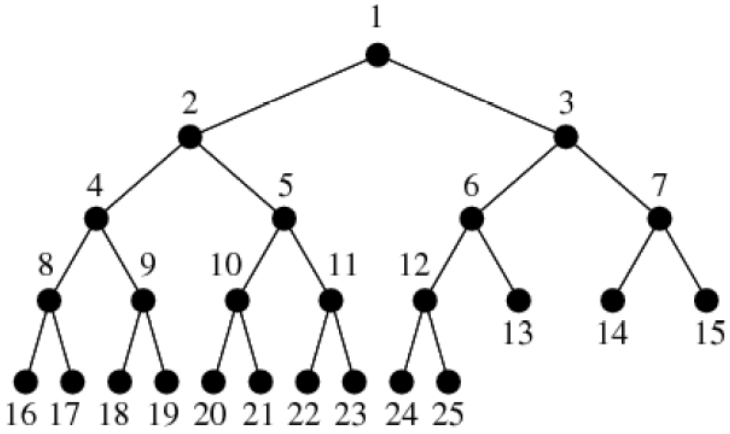
- $2^{h-1} \leq n \leq 2^h - 1$
- $\log_2 (n+1) \leq h \leq \log_2 n + 1$
- $h = \lfloor \log_2 n \rfloor + 1$

corolário: Numa árvore binária **completa** com n vértices, o comprimento máximo de um caminho desde a raiz até qualquer vértice é $O(\log_2 n)$.

propriedade: Uma árvore binária **completa** com n vértices é univocamente representável (por níveis) num vector de n elementos.



Assim, os vértices da árvore podem ser numerados:



❒ Amontoados Binários *(heaps)*

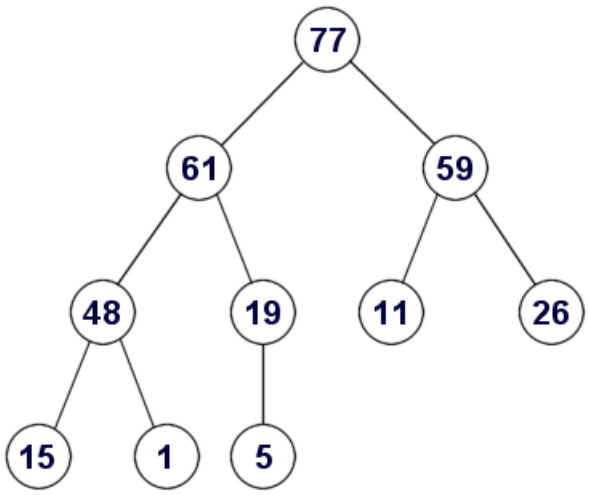
definição: Os elementos do vector x[1..n] dizem-se **amontoados** (formam um ***heap***) quando,

$$\forall\ k \in [2\ ..\ n] \Longrightarrow x[k\ \mathbf{div}\ 2] \geq x[k]$$

por exemplo,

	1	2	3	4	5	6	7	8	9	10
x	**77**	**61**	**59**	**48**	**19**	**11**	**26**	**15**	**1**	**5**

Representação sequencial de uma árvore binária completa.



Onde:

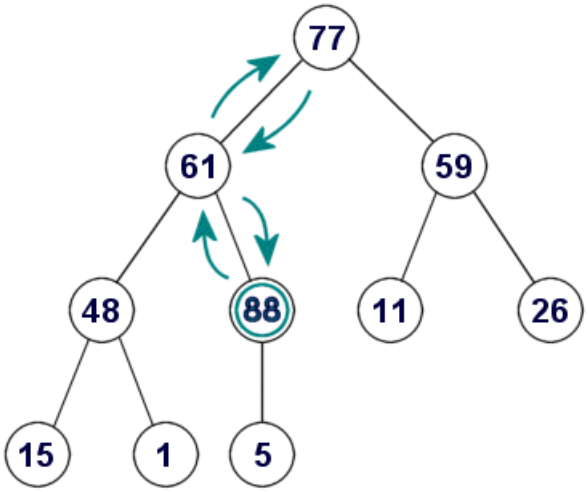
- $\forall\ k \in [1\ ..\ n\ \mathbf{div}\ 2] \Longrightarrow \quad x[k] \geq x[2k]$
  $x[k] \geq x[2k + 1]$
- x[k] é pai de x[2k] e de x[2k + 1]
- x[1] é o maior elemento do vector
- Tratando-se de uma árvore binária completa, as operações são $O(h) = O(\log_2 n)$.

Promoção

- Existe um único vértice k : x[k] > x[k **div** 2].
- Ir trocando com o pai, enquanto for preciso.
- Algoritmo:

enquanto x[k] > x[k **div** 2] e k > 1

x[k] ↔ x[k **div** 2]
k ← k **div** 2

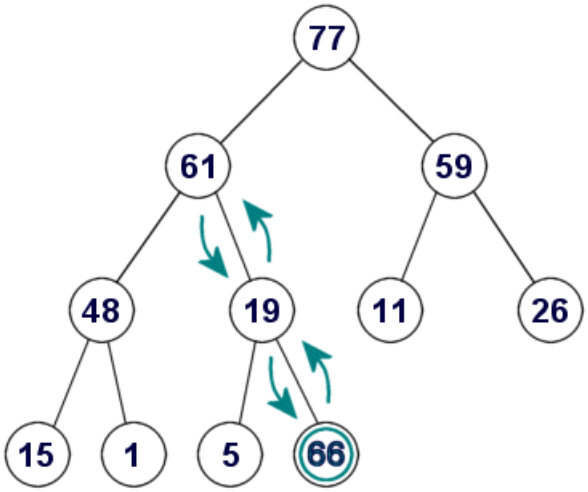


Inserção

- Inserir um novo valor em x[1..n].
- Juntar no fim e promover
- Algoritmo:

n ← n + 1
x[n] ← novo

promover(n)



Despromoção

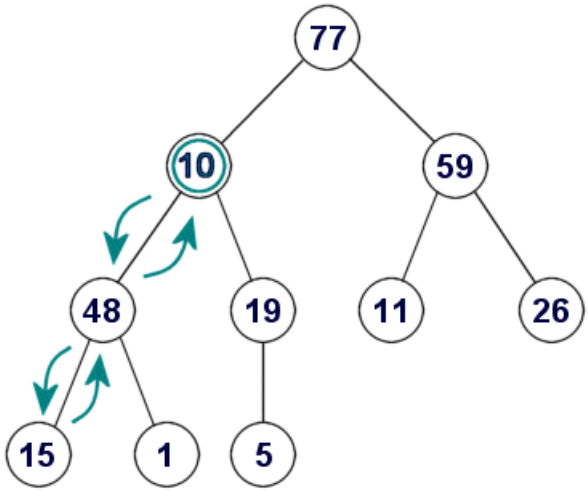
- Um único vértice $k : x[k] < x[2k]$ ou $x[k] < x[2k + 1]$.
- Ir trocando com o maior filho.
- Algoritmo:

enquanto $2k \leq n$

$j \leftarrow$ indice do max{ x[2k], x[2k+1] }

se $x[k] \geq x[j]$ parar (!)

$x[k] \leftrightarrow x[j]$
$k \leftarrow j$



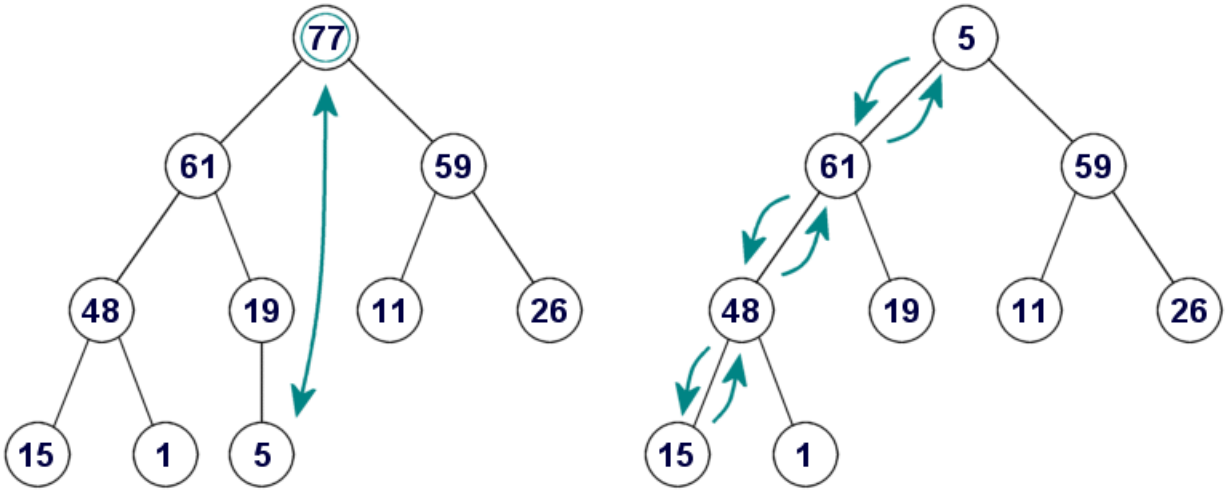
Remoção do Máximo

- Remover o elemento x[1].
- Trocar com o último e despromover.

- Algoritmo:

  x[1] ↔ x[n]
  n ← n - 1

  despromover(1)

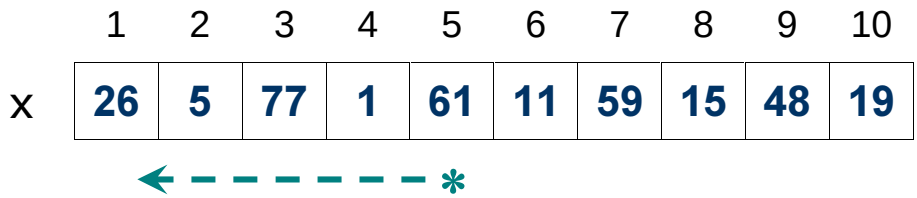


➡ Outras Operações / exercícios :

- Remover um elemento arbitrário.
- Alterar o valor de um elemento.
- Juntar dois *heaps*.
- Considerámos o caso do ***heap máximo***, onde o **maior** elemento do vector é o primeiro. Defina e analise o caso do ***heap mínimo***, onde o primeiro elemento é o **menor**.

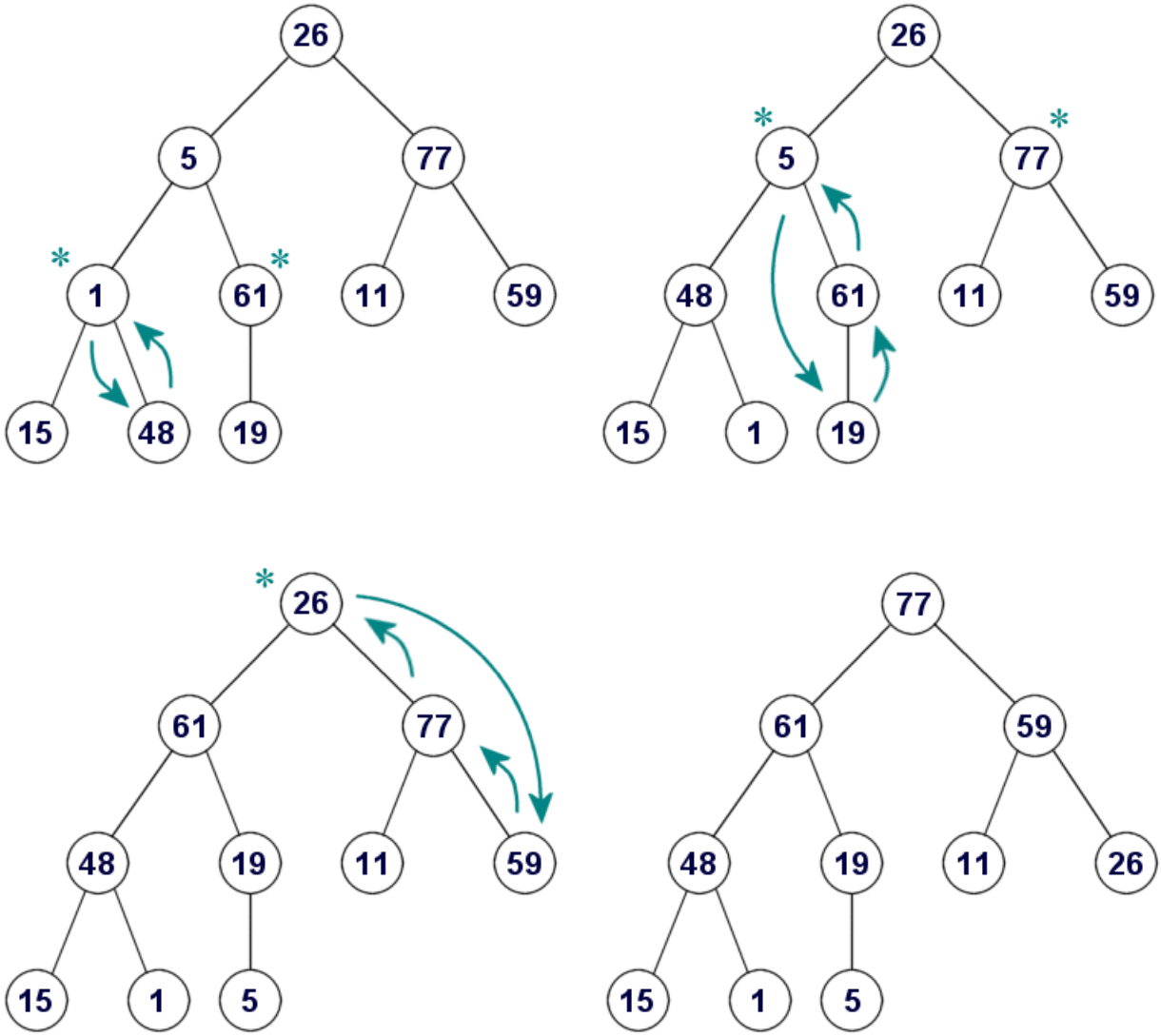
Construção de um *heap*

- Amontoar os elementos de um dado vector.



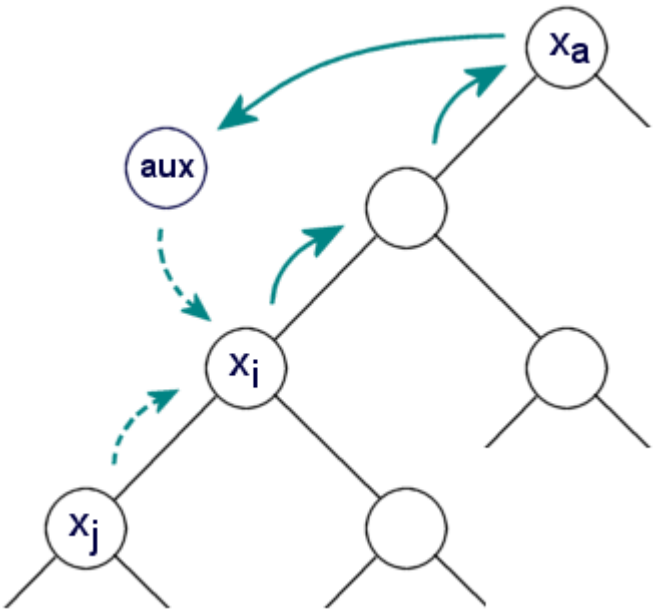
- Algoritmo:

para cada pai k desde n **div** 2 até 1
amontoar sub-árvore [k .. n]



- amontoar sub-árvore [a .. b]

```
i ← a          { pai da sub-árvore }
j ← 2 i        { e o seu primeiro filho }

aux ← x[i]  { guarda valor do pai }

enquanto j ≤ b

        j ←  indice do max{ x[j], x[j+1] }
             { o maior filho }

        se aux ≥ x[j]  parar (!)
        { encontrada localização do pai }

        x[i] ← x[j]
        { filho maior promovido a pai }

        i ← j       { novo pai }
        j ← 2 i     { e seu primeiro filho }

x[i] ← aux { recolocado pai da sub-árvore }
```



```
procedure AmontoarSubArvore ( a, b : indice );
var  i, j : indice;
     aux : elemento;
     achou : boolean;

begin  i := a;          { pai da sub-árvore }
       j := 2 * i;      { e o seu primeiro filho }

       aux := x[i];     { guarda valor do pai }

       achou := false;
       while  not achou and (j <= b) do
              begin  if j < b
                     then  if x[j] < x[j+1]
                           then j := j+1;       { o maior filho }

                     if aux >= x[j]
                     then  achou:= true
                           { encontrada localização do pai }

                     else  begin  { ainda não }
                                  x[i] := x[j];
                                  { filho maior promovido a pai }
                                  i := j;       { novo pai }
                                  j := 2 * i    { e seu primeiro filho }
                           end
              end;

       x[i] := aux       { recolocado pai da sub-árvore }

end; { AmontoarSubArvore }
```

- O procedimento AmontoarSubArvore consiste numa descida ao longo de uma árvore binária completa, cuja altura máxima é $O(\log_2 n)$.
- Note que, no pior dos casos: (j := 1) (j := 2 * i) até (j = n).

- O processo completo da **construção do *heap***, será então:

{ sub-árvores x[n **div** 2 + 1 .. n] amontoadas } { folhas }

for k := n **div** 2 **downto** 1 **do**
{ sub-árvores x[k+1 .. n] amontoadas }
AmontoarSubArvore(k, n)
{ sub-árvores x[k .. n] amontoadas } ;

{ árvore x[1 .. n] amontoada }

- Portanto, a complexidade da operação completa da construção do *heap* é $O(n \log_2 n)$.

➡ Complexidades:

vector	inserção	remoção do máximo	pesquisa do máximo
não ordenado	$O(1)$	$O(n)$	$O(n)$
ordenado	$O(n)$	$O(1)$	$O(1)$
heap binário	$O(\log n)$	$O(\log n)$	$O(1)$

➡ Aplicações dos *heaps* binários:

- Um dos melhores métodos de ordenamento.
- Filas de Prioridade.
- ...

❒ *HeapSort*

- A partir de um vector amontoado, como colocar os seus elementos por **ordem não decrescente**?

	1	2	3	4	5	6	7	8	9	10
x	**77**	**61**	**59**	**48**	**19**	**11**	**26**	**15**	**1**	**5**

- O primeiro elemento é o maior: x[1] ↔ x[10]
  AmontoarSubArvore(1, 9)

	1	2	3	4	5	6	7	8	9	10
x	**61**	**48**	**59**	**15**	**19**	**11**	**26**	**5**	**1**	**77**

- x[1] ↔ x[9] ; AmontoarSubArvore(1, 8)

	1	2	3	4	5	6	7	8	9	10
x	**59**	**48**	**26**	**15**	**19**	**11**	**1**	**5**	**61**	**77**

- x[1] ↔ x[8] ; AmontoarSubArvore(1, 7)

	1	2	3	4	5	6	7	8	9	10
x	**48**	**19**	**26**	**15**	**5**	**11**	**1**	**59**	**61**	**77**

- …
- x[1] ↔ x[2] ; AmontoarSubArvore(1, 1)

	1	2	3	4	5	6	7	8	9	10
x	**1**	**5**	**11**	**15**	**19**	**26**	**48**	**59**	**61**	**77**

```
procedure HeapSort ( var x : vector; n : indice );
var   k : indice;
      aux : elemento;

begin  for   k := n div 2  downto  1 do
             AmontoarSubArvore(k, n);
       { x[1 .. n] amontoado  ∧  x[ ] ordenado }

       for   k := n-1 downto  1 do
             begin  { x[1 .. k+1] amontoado  ∧  x[k+2 .. n] ordenado }

                    aux := x[1];
                    x[1] := x[k+1];
                    x[k+1] := aux;
                    { x[k+1 .. n] ordenado }

                    AmontoarSubArvore(1, k)
                    { x[1 .. k] amontoado  ∧  x[k+1 .. n] ordenado }
             end;
       { x[1] amontoado  ∧  x[2 .. n] ordenado }
       { x[1 .. n] ordenado }

end; { HeapSort }
```

- O primeiro ciclo, da construção inicial do ***heap***, é $O(n \log_2 n)$.
- O segundo ciclo é sequencial ao primeiro e tem complexidade análoga.
- Portanto o algoritmo ***HeapSort*** é $O(n \log_2 n)$, o limite mínimo para os métodos de ordenamento baseados em trocas.
- Note-se que não existe um Pior Caso $O(n^2)$, como acontece no ***QuickSort***.

❒ Filas de Prioridade

- Muitas vezes é necessária uma estrutura de dados, semelhante à Fila, mas com um sistema de prioridades associado aos seus elementos.
- O "primeiro" elemento é aquele cuja prioridade é máxima e, em caso de igualdade, deverá funcionar como uma Fila.
- De forma eficiente, é necessário conhecer o "primeiro" elemento, removê-lo e reorganizar a estrutura.

➡ Aplicações das Filas de Prioridade:

- Simulação Controlada pelos Eventos.
- Computação Numérica.
- Compressão de Dados. {Códigos de Huffman}
- Pesquisa em Grafos. {Algoritmos de Dijkstra e de Prim}
- Algoritmos em Teoria dos Números.
- Inteligência Artificial.
- Estatística.
- Sistemas Operativos.
- Optimização Discreta.
- Filtros de *spam*.
- ...

➡ Operações Básicas no TAD Fila de Prioridade:

nula(FP)	{ verificar se a FP está vazia }
construir(n, FP)	{ construir uma FP a partir de n elementos }
maximo(FP)	{ o elemento de valor máximo }
remmaximo(FP)	{ remover o máximo e reordenar }
comprimento(FP)	{ o número de elementos }
inserir(e, FP)	{ inserir um novo elemento e reordenar }

- Como implementar uma Fila de Prioridade?

Possíveis Implementações de uma Fila de Prioridade:

	inserção	remoção do máximo	pesquisa do máximo
vector não ordenado	$O(1)$	$O(n)$	$O(n)$
vector ordenado	$O(n)$	$O(1)$	$O(1)$
lista não ordenada	$O(1)$	$O(n)$	$O(n)$
lista ordenada	$O(n)$	$O(1)$	$O(1)$
árvore equilibrada	$O(\log n)$	$O(\log n)$	$O(\log n)$
heap binário	$O(\log n)$	$O(\log n)$	$O(1)$

- O ***heap* binário** é portanto uma estrutura adequada à implementação das Filas de Prioridade, permitindo operações de complexidade sub-linear, sem utilização explícita de ponteiros.