Universidade de Trás-os-Montes e Alto Douro

Algoritmia
2017/2018

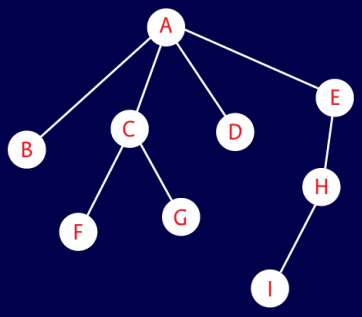
Pedro Melo-Pinto
(pmelo@utad.pt)

Algoritmia

{ *árvores binárias* }

Estruturas não lineares de dados

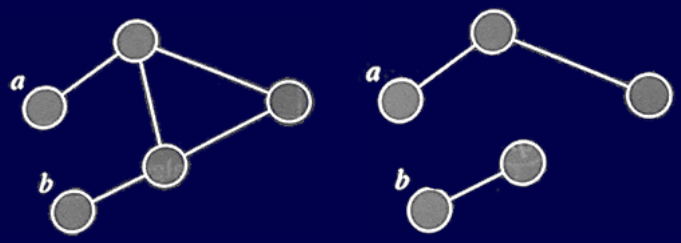
Árvores



conjunto de dados

relações entre os dados ⟶ para ir de um qualquer vértice para qualquer outro vértice, existe um e um só caminho

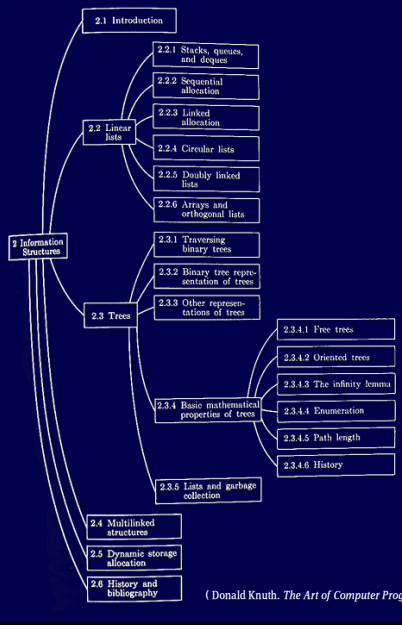
Árvores



conjunto de dados

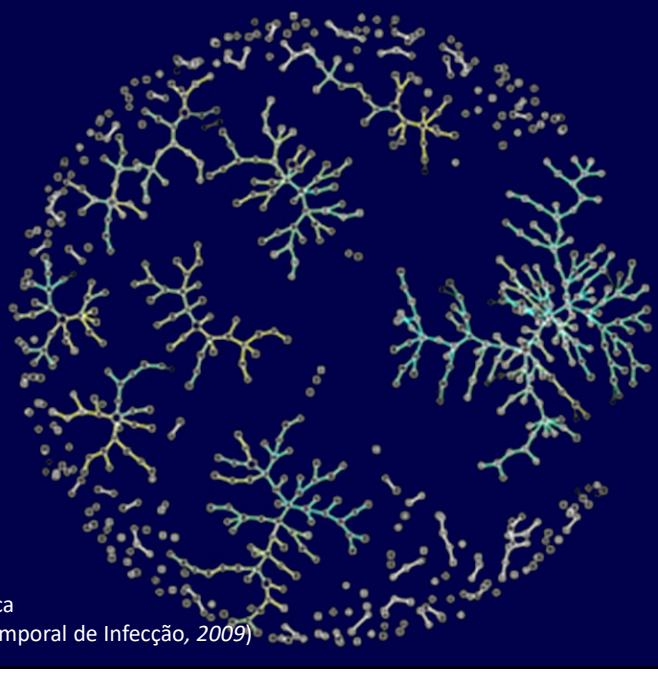
relações entre os dados ⟶ para ir de um qualquer vértice para qualquer outro vértice, existe um e um só caminho

Árvores



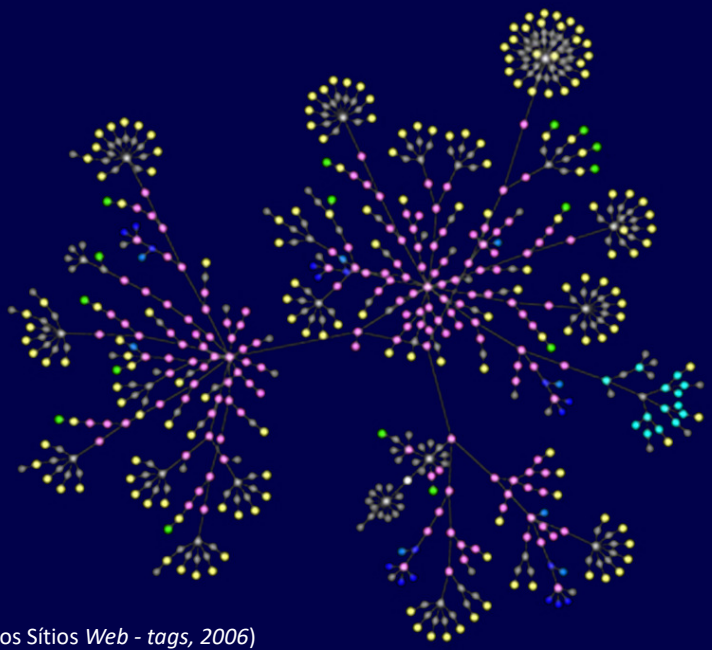
(Donald Knuth. *The Art of Computer Programming, 2nd edition.*)

Transportes
(Transportes Ferroviários Hankyu, Japão. *Mapa, 2007*)



Saúde Pública
(Evolução temporal de Infecção, *2009*)

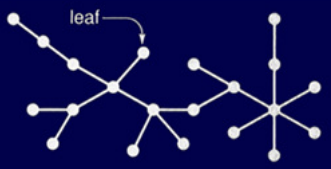
msn.com



Internet
(Estrutura dos Sítios *Web - tags, 2006*)

Árvores

árvores (*free trees*)
árvores com raiz (*rooted trees*)
árvores ordenadas (*ordered trees*)
árvores (n-árias e) binárias



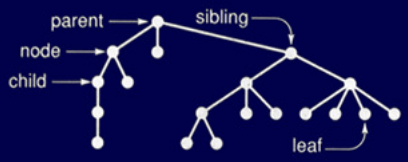
(Robert Sedgewick. *Algorithms in C, 2nd edition.*)

conjunto não vazio de vértices e ligações que obedece a certas condições

existe exactamente um caminho a ligar dois vértices da árvore

Árvores

árvores (*free trees*)
árvores com raiz (*rooted trees*)
árvores ordenadas (*ordered trees*)
árvores (n-árias) e binárias



(Robert Sedgewick. *Algorithms in C, 2nd edition.*)

é uma árvore com um nó designado raiz da árvore

existe exactamente um caminho entre a raiz e qualquer outro nó da árvore

não existe sentido explícito nas ligações

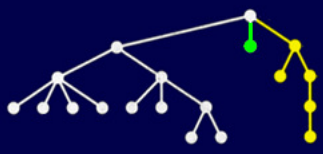
cada nó (excepto a raiz) tem um nó acima, designado por pai (*parent*)

os nós imediatamente abaixo são os filhos (*children*; *siblings*)

os nós sem filhos chamam-se folhas

Árvores

árvores (*free trees*)
árvores com raiz (*rooted trees*)
árvores ordenadas (*ordered trees*)
árvores (n-árias) e binárias



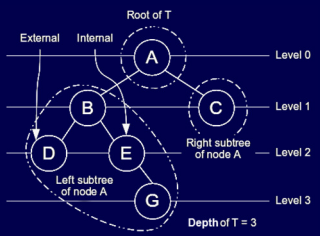
(Robert Sedgewick. *Algorithms in C, 2nd edition.*)

é uma árvore com raiz

a ordem dos filhos em cada nó tem significado

Árvores

árvores (*free trees*)
árvores com raiz (*rooted trees*)
árvores ordenadas (*ordered trees*)
árvores (n-árias e) binárias



Tree T

(https://gsourcecode.wordpress.com/2012/02/12/fs-bt-int/)

cada nó possui duas sub-árvores

os nós podem ser externos (sem filhos), ou internos (com um ou dois filhos)

os filhos dos nós internos designam-se por filho esquerdo e direito respectivamente

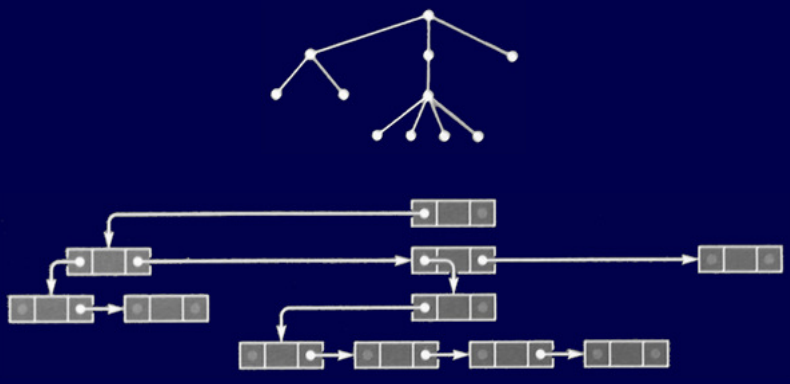
uma árvore binária cuja raiz é nula (vazia) é uma árvore binária vazia

uma árvore binária é uma árvore ordenada

Árvores

metodologias de implementação :

utilização de listas duplamente encadeadas

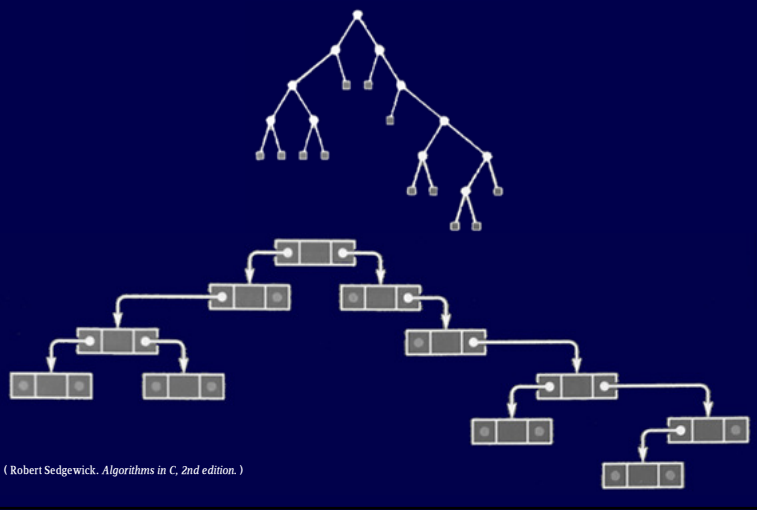


(Robert Sedgewick. *Algorithms in C, 2nd edition.*)

Árvores

metodologias de implementação :

utilização de listas duplamente encadeadas



(Robert Sedgewick. *Algorithms in C, 2nd edition.*)

Árvores (binárias)

operações típicas :

inserção da raiz

inserção de vértices (do filho esquerdo ou direito de uma folha)

eliminação de um vértice (folha)

percorrer a árvore

união de duas árvores

algoritmos de ordenação

procurar determinado elemento numa árvore (pode ser um máximo ou mínimo)

Árvores (binárias)

(algumas) aplicações :

sistemas operativos - sistema de alocação de ficheiros

processos de ordenação

processos de pesquisa

conectividade

caminhos mais curtos

Árvores binárias

algumas definições e propriedades (matemáticas)

O nível de cada nó é maior em um ao nível do seu pai. e é o nº de ligações desde a raiz até si

O nível da raíz é 0 (zero).

A profundidade de uma árvore binária é o nível máximo dos seus nós.

O comprimento do caminho de uma árvore binária é a soma do nível de todos os seus nós.

Uma árvore binária com N nós internos tem, no máximo, N+1 nós externos (folhas).

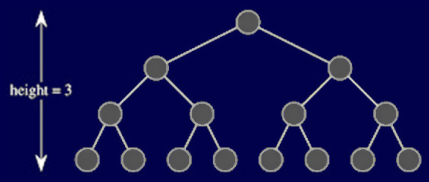
Uma árvore binária com N nós internos tem, no máximo, 2N ligações: N - 1 ligações para nós internos e (no máximo) N + 1 ligações para nós externos.

A profundidade de uma árvore binária com **N** (**N** > 0) nós é, pelo menos, $\lfloor \log_2 \mathbf{N} \rfloor$ e, no máximo **N**-1.

Uma árvore binária (não vazia) com profundidade h tem, no mínimo, h + 1 nós e, no máximo $2^{h+1} - 1$.

Árvores binárias

algumas definições e propriedades (matemáticas)



(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

Uma árvore binária cheia (perfect BT) é aquela em que todas as folhas têm o mesmo nível e todos os nós internos têm dois filhos.

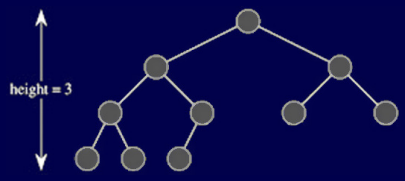
Uma árvore binária cheia tem $2^{h+1} - 1$ nós, em que h é a profundidade da árvore.

Uma árvore binária cheia tem 2^{h} nós externos, em que h é a profundidade da árvore.

Uma árvore binária cheia tem $2^{h} - 1$ nós internos, em que h é a profundidade da árvore.

Árvores binárias

algumas definições e propriedades (matemáticas)



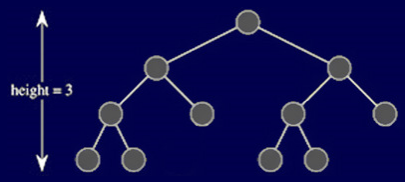
(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

Uma árvore binária completa (complete BT) é uma árvore binária em que todos os níveis, excepto (eventualmente) o último, estão completos, e em que todos os nós estão tão à esquerda quanto possível.

Uma árvore binária completa tem, no máximo, $2^{h+1} - 1$ nós e, no mínimo 2^{h} , em que h é a profundidade da árvore.

Árvores binárias

algumas definições e propriedades (matemáticas)



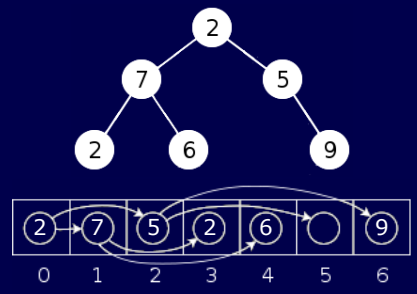
(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

Uma árvore binária própria (full BT; proper BT) é uma árvore binária em que cada nó tem 0 ou 2 filhos.

Uma árvore binária própria tem, no máximo, $2^{h+1} - 1$ nós e, no mínimo $2^h + 1$, em que h é a profundidade da árvore.

Árvores binárias

implementação (alternativa) de árvores binárias – através de vectores



Para o caso de árvores binárias completas esta metodologia de implementação faz um uso eficaz dos recursos de memória (não desperdiça espaço).

Para um nó cujo índice seja *i*, os seus filhos estão nas posições com índices $2i + 1$ (filho da esquerda) and $2i + 2$ (filho da direita).

Para um nó cujo índice seja *i*, o seu pai, a existir, está na posição com índice $\lfloor (i - 1)/2 \rfloor$.

Árvores binárias: *algoritmos de percurso*

o objectivo de percorrer uma árvore é visitar (e actuar sobre) cada nó da árvore

o processo (ao contrário do que acontecia em listas) deixou de ser único

em árvores binárias existem 4 processos para percorrer uma árvore :

- método *preorder*
- método *inorder*
- método *postorder*

} depth-first (em profundidade)

- percorrer por nível (*level traversal*) — breadth-first (em abrangência)

estes métodos variam na ordem como se processam as "visitas" (recursivas) em cada nó:

(*preorder*) 1. visita-se o filho da esquerda
2. visita-se o filho da esquerda
3. visita-se o filho da direita

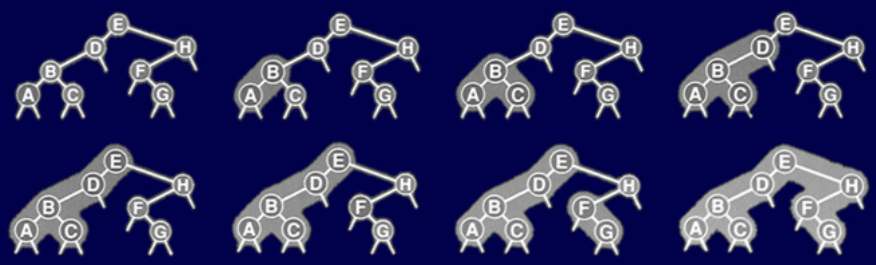
o quarto método percorre a árvore por nível, da esquerda para a direita

Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          print key[x]
4          INORDER-TREE-WALK(right[x])
```

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)



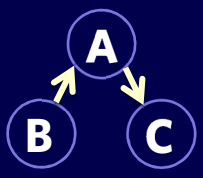
(Robert Sedgewick. *Algorithms in C, 2nd edition.*)

A B C D E F G H

Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          print key[x]
4          INORDER-TREE-WALK(right[x])
```



regra:

A: (nó) pai de B e C, é a raíz destes três nós.
B: (nó) filho da esquerda de A
C: (nó) filho da direita de A
Para cada nó "visita" (no caso acima *print*) "filho esquerda→raiz→ filho da direita"

Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

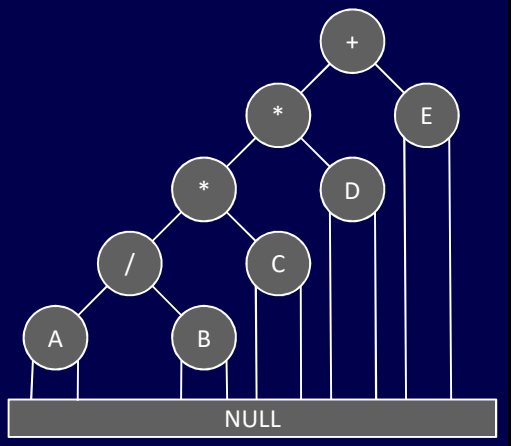
```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          print key[x]
4          INORDER-TREE-WALK(right[x])
```

```
void inorder(tree_pointer ptr){
    if(ptr){
        inorder(ptr->left_child);
        printf("%d", ptr->data);
        inorder(ptr->right_child);
    }
}
```

Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

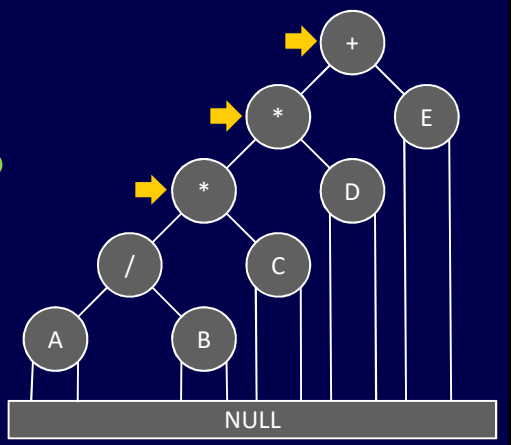
```
void inorder(tree_pointer ptr){
   if(ptr){
      inorder(ptr->left_child);
      printf("%d", ptr->data);
      inorder(ptr->right_child);
   }
}
```



Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

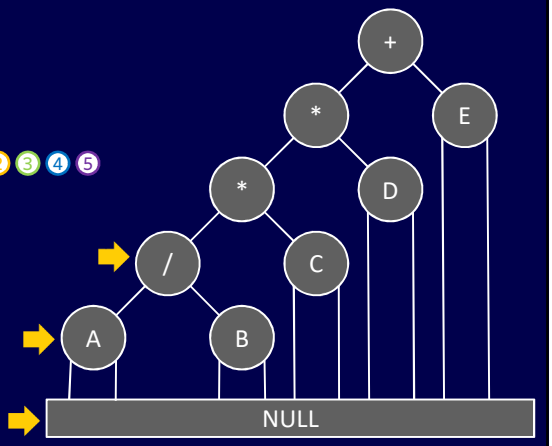
```
void inorder(tree_pointer ptr){ 3
   if(ptr){ 3
      inorder(ptr->left_child); 1 2 3
      printf("%d", ptr->data);
      inorder(ptr->right_child);
   }
}
```



Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

```
void inorder(tree_pointer ptr){ 6
   if(ptr){ 6
      inorder(ptr->left_child); 1 2 3 4 5
      printf("%d", ptr->data);
      inorder(ptr->right_child);
   }
}
```

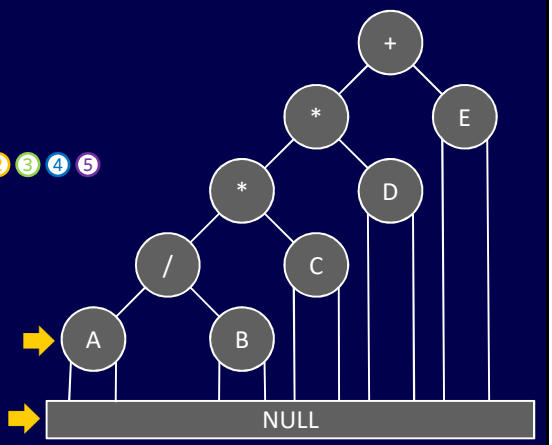


Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

```
void inorder(tree_pointer ptr){ 6
   if(ptr){ 6
      inorder(ptr->left_child); 1 2 3 4 5
      printf("%d", ptr->data); 5
      inorder(ptr->right_child); 5
   }
}
```

A

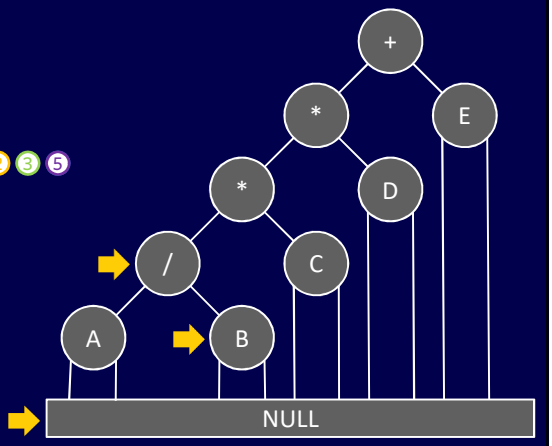


Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

```
void inorder(tree_pointer ptr){ 6
   if(ptr){ 6
      inorder(ptr->left_child); 1 2 3 5
      printf("%d", ptr->data); 4
      inorder(ptr->right_child); 4
   }
}
```

A /

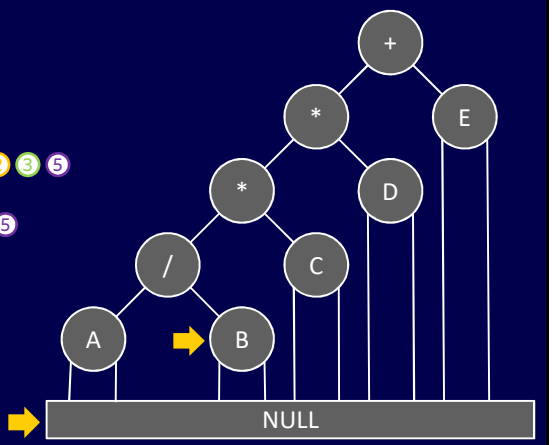


Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

```
void inorder(tree_pointer ptr){ 6
   if(ptr){ 6
      inorder(ptr->left_child); 1 2 3 5
      printf("%d", ptr->data); 5
      inorder(ptr->right_child); 4 5
   }
}
```

A / B

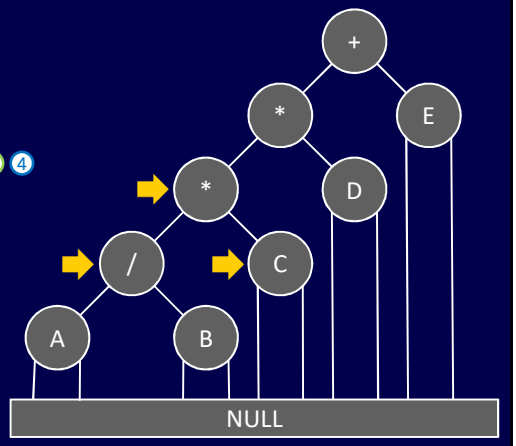


Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

```
void inorder(tree_pointer ptr){ 4
   if(ptr){ 4
      inorder(ptr->left_child); 1 2 3 4
      printf("%d", ptr->data); 3
      inorder(ptr->right_child); 3
   }
}
```

A / B *

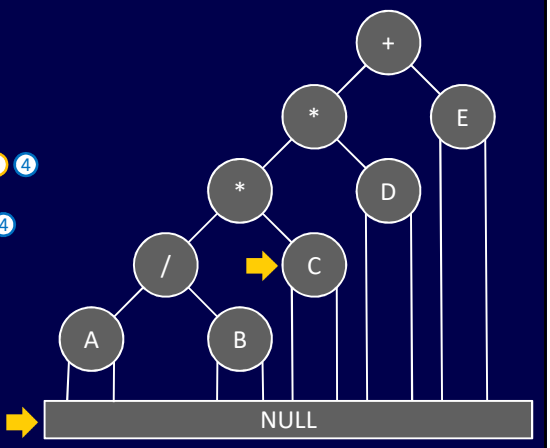


Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

```
void inorder(tree_pointer ptr){ 5
   if(ptr){ 5
      inorder(ptr->left_child); 1 2 4
      printf("%d", ptr->data); 4
      inorder(ptr->right_child); 3 4
   }
}
```

A / B * C

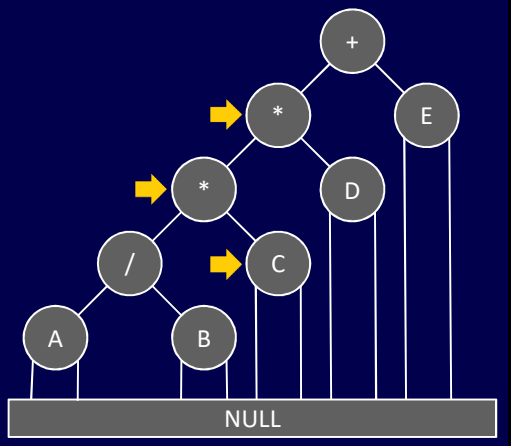


Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

```
void inorder(tree_pointer ptr){
    if(ptr){
        inorder(ptr->left_child);  1 2
        printf("%d", ptr->data);   2
        inorder(ptr->right_child); 2 4
    }
}
```

A / B * C *

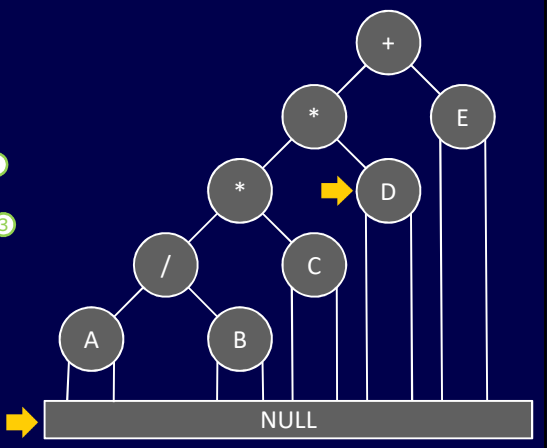


Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

```
void inorder(tree_pointer ptr){ 4
    if(ptr){ 4
        inorder(ptr->left_child);  1 3
        printf("%d", ptr->data);   3
        inorder(ptr->right_child); 2 3
    }
}
```

A / B * C * D

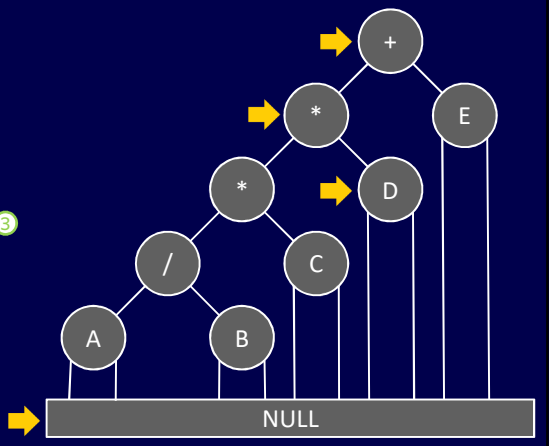


Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

```
void inorder(tree_pointer ptr){ 4
   if(ptr){ 4
      inorder(ptr->left_child); 1
      printf("%d", ptr->data); 1
      inorder(ptr->right_child); 1 3
   }
}
```

A / B * C * D +

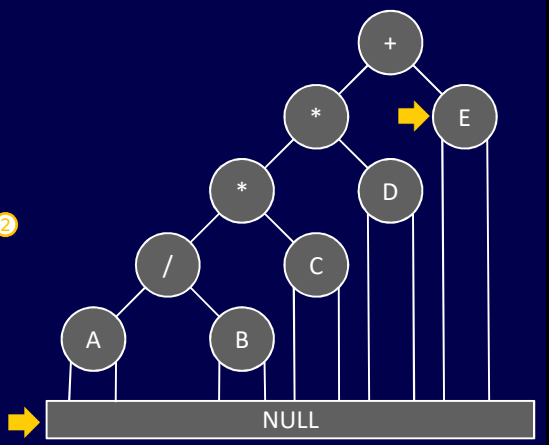


Árvores binárias: *algoritmos de percurso*

algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

```
void inorder(tree_pointer ptr){ 3
   if(ptr){ 3
      inorder(ptr->left_child); 2
      printf("%d", ptr->data); 2
      inorder(ptr->right_child); 1 2
   }
}
```

A / B * C * D + E

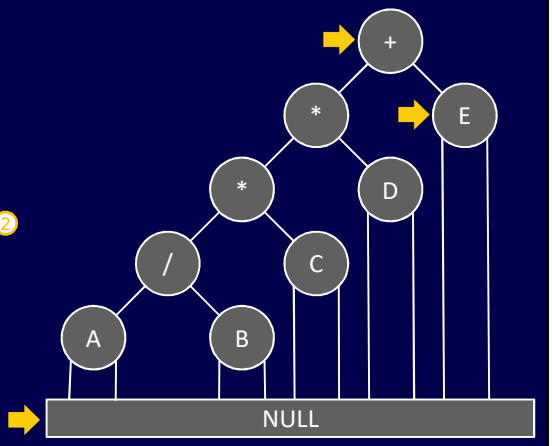


Árvores binárias: *algoritmos de percurso*

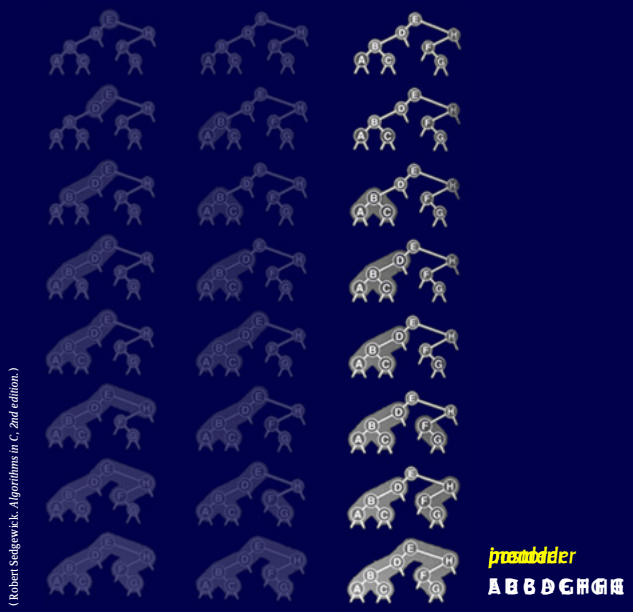
algoritmo da operação **PERCORRER ÁRVORE** utilizando o método *inorder*

```
void inorder(tree_pointer ptr){ 3
    if(ptr){ 3
        inorder(ptr->left_child);
        printf("%d", ptr->data);
        inorder(ptr->right_child); 1 2
    }
}
```

A / B * C * D + E



Árvores binárias: *algoritmos de percurso*



(Robert Sedgewick. *Algorithms in C, 2nd edition.*)

Árvores binárias: *algoritmos de percurso*

Esta operação tem uma complexidade do tempo de execução O(n).

$T(n) = T(k) + T(n - k - 1) + c$ - *k* é o nº de nós num dos lados da árvore e *n-k-1* no outro

Analisemos os dois casos extremos

Árvores binárias: *algoritmos de percurso*

Esta operação tem uma complexidade do tempo de execução O(n).

$T(n) = T(k) + T(n - k - 1) + c$ - *k* é o nº de nós num dos lados da árvore e *n-k-1* no outro

Caso 1 (um dos lados da árvore está vazio, k=0):

$T(n) = T(0) + T(n-1) + c$
$T(n) = 2T(0) + T(n-2) + 2c$
...
$T(n) = nT(0) + (n)c$

$T(0) = d$ (percorrer uma árvore vazia leva tempo constante - d)

$T(n) = n(c+d)$
$T(n) = \Theta(n)$

Árvores binárias: *algoritmos de percurso*

Esta operação tem uma complexidade do tempo de execução O(n).

$T(n) = T(k) + T(n - k - 1) + c$ - *k* é o nº de nós num dos lados da árvore e *n-k-1* no outro

Caso 2 (ambos os lados da árvore têm o mesmo número de nós):
$T(n) = 2T(\lfloor n/2 \rfloor) + c$

Master Theorem: $T(n) = aT(n/b) + f(n)$

No nosso caso
a=2, b=2, f(n) = c → $n^{\log_b a} = n^{\log_2 2} = n$

Solução
$T(n) = \Theta(n)$

Análise de Recorrências

Master Theorem

$T(n) = a \times T\left(\frac{n}{b}\right) + f(n)$, where $\frac{n}{b}$ can either be interpreted as $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$

1. if $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ *for some* $\epsilon > 0$
   then $T(n) = \Theta\left(n^{\log_b a}\right)$

2. if $f(n) = \Theta\left(n^{\log_b a}\right)$
   then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$

3. if $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ *for some* $\epsilon > 0$
   if $af\left(\frac{n}{b}\right) \leq cf(n)$ *for* $c < 1$ *and all sufficiently large* n
   then $T(n) = \Theta(f(n))$

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

Árvores binárias: *algoritmos de percurso*

Esta operação tem uma complexidade do tempo de execução O(n).

$T(n) = T(k) + T(n - k - 1) + c$ - *k* é o nº de nós num dos lados da árvore e *n-k-1* no outro

Caso 2 (ambos os lados da árvore têm o mesmo número de nós):
$T(n) = 2T(\lfloor n/2 \rfloor) + c$

Master Theorem: $T(n) = aT(n/b) + f(n)$

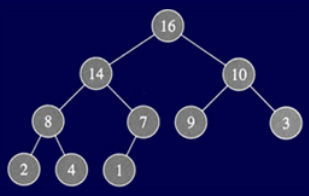
No nosso caso

1. $c = O(n^{1-\epsilon}) = O(1), para\ \epsilon = 1$ **OK**
2. $c = \Theta(n)$ **FALSO**
3. $c = \Omega(n^{1+\epsilon}) = \Omega(n^2), para\ \epsilon = 1$ **FALSO**

Solução
$T(n) = \Theta(n)$

Árvores binárias : *heaps*

Árvores binárias: *heaps*



(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

é uma árvore binária completa que obedece a uma condição

existem dois tipos de *heaps*: *max-heaps* e *min-heaps*

numa *min-heap* o valor de um vértice não pode ser inferior ao do seu pai

o valor mínimo da árvore encontra-se na raiz

numa *max-heap* o valor de um vértice não pode ser superior ao do seu pai

o valor máximo da árvore encontra-se na raiz

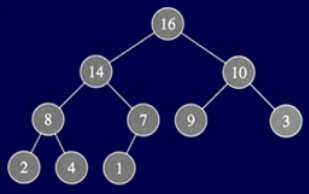
Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*



(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

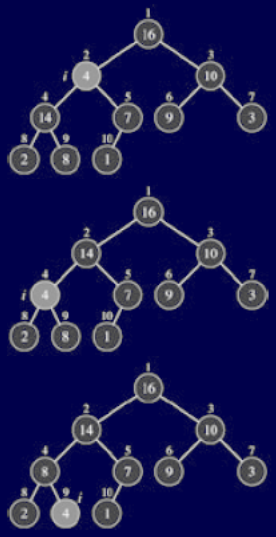
Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*



```
TOP-DOWN-MAX-HEAPIFY (A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ HEAP-SIZE(A) and A[l] > A[i]
4     then largest ← l
5     else largest ← i
6  if r ≤ HEAP-SIZE[A] and A[r] > A[largest]
7     then largest ← r
8  if largest ≠ i
9     then exchange A[i] ↔ A[largest]
10         TOP-DOWN-MAX-HEAPIFY (A, largest)
```

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*

Top-Down Heapify (sink)

```
TOP-DOWN-MAX-HEAPIFY (A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ HEAP-SIZE(A) and A[l] > A[i]
4     then largest ← l
5     else largest ← i
6  if r ≤ HEAP-SIZE[A] and A[r] > A[largest]
7     then largest ← r
8  if largest ≠ i
9     then exchange A[i] ↔ A[largest]
10         TOP-DOWN-MAX-HEAPIFY (A, largest)
```

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

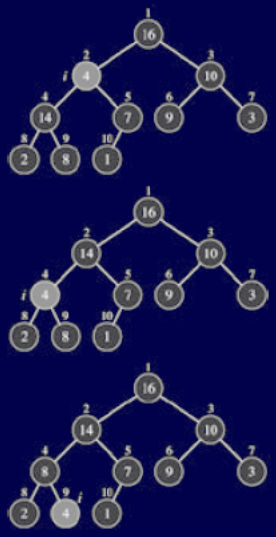
Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*



Top-Down Heapify (sink)

Análise:
$T(n) \leq T(2n/3) + \Theta(1)$
$T(n) = O(\lg n)$

Para um nó de nível h
$T(h) = O(h)$

```
7     then largest ← r
8  if largest ≠ i
9     then exchange A[i] ↔ A[largest]
10         TOP-DOWN-MAX-HEAPIFY (A, largest)
```

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*

```
BOTTOM-UP-MAX-HEAPIFY (A, i)
1  while i > 1 and A[PARENT(i)] < A[i]
2     do exchange A[i] ↔ A[PARENT(i)]
3        i ← PARENT(i)
```

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*

Bottom-Up Heapify (swim)


```
BOTTOM-UP-MAX-HEAPIFY (A, i)
1 while i > 1 and A[PARENT(i)] < A[i]
2    do exchange A[i] ↔ A[PARENT(i)]
3       i ← PARENT(i)
```

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*

Bottom-Up Heapify (swim)

Análise:
$T(n) \leq T(2n/3) + \Theta(1)$
$T(n) = O(\lg n)$

Para um nó de nível h
$T(h) = O(h)$


```
BOTTOM-UP-MAX-HEAPIFY (A, i)
1 while i > 1 and A[PARENT(i)] < A[i]
2    do exchange A[i] ↔ A[PARENT(i)]
3       i ← PARENT(i)
```

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

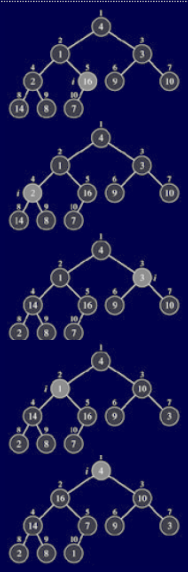
Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*



vector original A

4	1	3	2	16	9	10	14	8	7
1	2	3	4	5	6	7	8	9	10

```
BUILD-MAX-HEAP (A)
1  HEAP-SIZE ← length(A)
2  for i ← ⌊ lenght(A)/2 ⌋downto 1
3     do  TOP-DOWN-MAX-HEAPIFY (A, i)
```

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

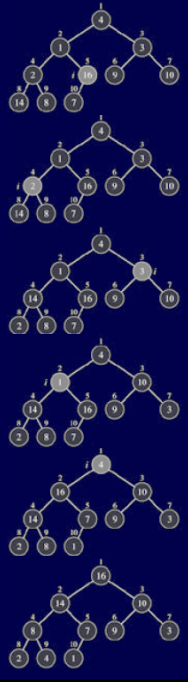
Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*



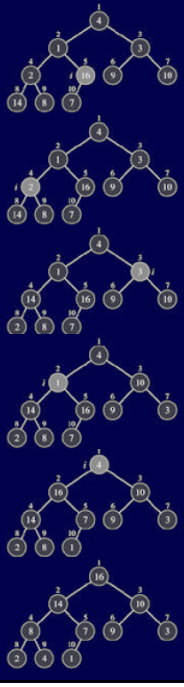
vector final A

16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

```
BUILD-MAX-HEAP (A)
1  HEAP-SIZE ← length(A)
2  for i ← ⌊ lenght(A)/2 ⌋downto 1
3     do  TOP-DOWN-MAX-HEAPIFY (A, i)
```

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

Árvores binárias: *heaps*



operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*

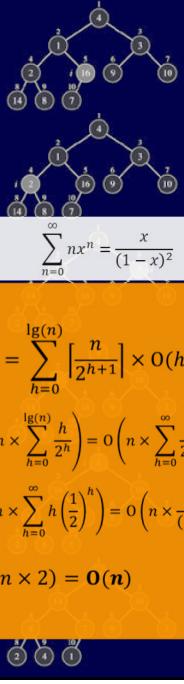
Análise:

Limite superior
Cada Max-Heapify demora O(lg n)
Há O(n) chamadas recursivas de Max-Heapify
T(n) = O(n lg n)

Atendendo a que T(h) = O(h) e que uma *heap* de n elementos tem, no máximo, $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nós, chegamos a um valor mais apertado do limite superior
T(n) = O(n)

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

Árvores binárias: *heaps*



operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*

$$\sum_{n=0}^{\infty} nx^n = \frac{x}{(1-x)^2}$$

Análise:

Limite superior
Cada Max-Heapify demora O(lg n)
Há O(n) chamadas recursivas de Max-Heapify
T(n) = O(n lg n)

Atendendo a que T(h) = O(h) e que uma *heap* de n elementos tem, no máximo, $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nós, chegamos a um valor mais apertado do limite superior
T(n) = O(n)

$$T(n) = \sum_{h=0}^{\lg(n)} \left\lceil \frac{n}{2^{h+1}} \right\rceil \times O(h)$$

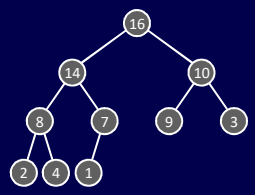
$$= O\left(n \times \sum_{h=0}^{\lg(n)} \frac{h}{2^h}\right) = O\left(n \times \sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$

$$= O\left(n \times \sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h\right) = O\left(n \times \frac{1/2}{(1-1/2)^2}\right)$$

$$= O(n \times 2) = \mathbf{O(n)}$$

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

Árvores binárias: *heaps*



operações típicas :

inicialização e manutenção

inserção e eliminação de nós

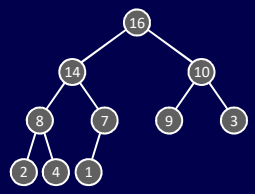
método de ordenação *heapsort*

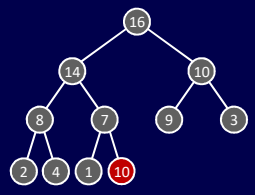
vector A

16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

```
INSERT (A, X)
1 A[HEAP-SIZE + 1] ← X
2 HEAP-SIZE ← HEAP-SIZE + 1
3 BOTTOM-UP-MAX-HEAPIFY (A, HEAP-SIZE)
```


Árvores binárias: *heaps*





operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*

vector A

16	14	10	8	7	9	3	2	4	1	10
1	2	3	4	5	6	7	8	9	10	11

```
INSERT (A, X)
1 A[HEAP-SIZE + 1] ← X
2 HEAP-SIZE ← HEAP-SIZE + 1
3 BOTTOM-UP-MAX-HEAPIFY (A, HEAP-SIZE)
```

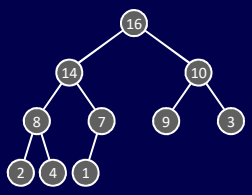
Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

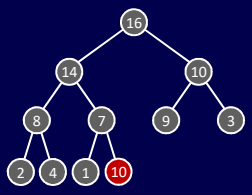
inserção e eliminação de nós

método de ordenação *heapsort*

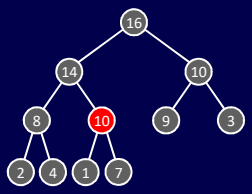


vector A

16	14	10	8	10	9	3	2	4	1	7
1	2	3	4	5	6	7	8	9	10	11




```
INSERT (A, X)
1 A[HEAP-SIZE + 1] ← X
2 HEAP-SIZE ← HEAP-SIZE + 1
3 BOTTOM-UP-MAX-HEAPIFY (A, HEAP-SIZE)
```



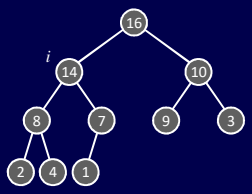
Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*



vector A

16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

```
REMOVE (A, i)
1  exchange A[i] ↔ A[HEAP-SIZE]
2  HEAP-SIZE ← HEAP-SIZE - 1
3  TOP-DOWN-MAX-HEAPIFY (A, i)
4  return A[HEAP-SIZE + 1]
```

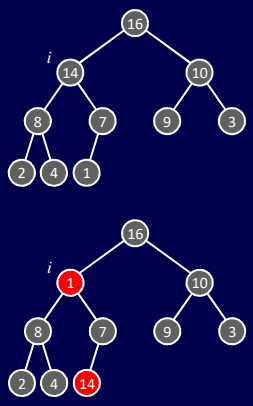
Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*



vector A

16	1	10	8	7	9	3	2	4	14
1	2	3	4	5	6	7	8	9	10

```
REMOVE (A, i)
1  exchange A[i] ↔ A[HEAP-SIZE]
2  HEAP-SIZE ← HEAP-SIZE - 1
3  TOP-DOWN-MAX-HEAPIFY (A, i)
4  return A[HEAP-SIZE + 1]
```

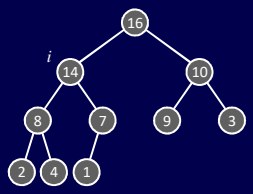

Árvores binárias: *heaps*

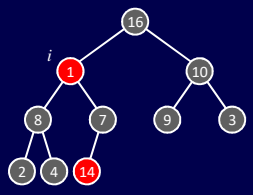
operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*

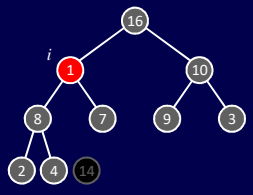




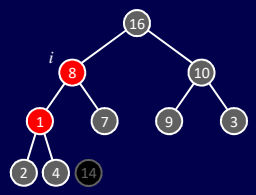
vector A

16	1	10	8	7	9	3	2	4	14
1	2	3	4	5	6	7	8	9	10

```
REMOVE (A, i)
1  exchange A[i] ↔ A[HEAP-SIZE]
2  HEAP-SIZE ← HEAP-SIZE - 1
3  TOP-DOWN-MAX-HEAPIFY (A, i)
4  return A[HEAP-SIZE + 1]
```



Árvores binárias: *heaps*



operações típicas :

inicialização e manutenção

inserção e eliminação de nós

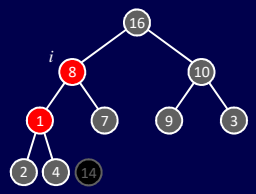
método de ordenação *heapsort*

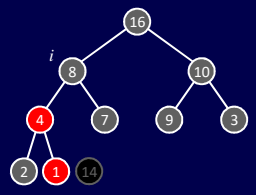
vector A

16	8	10	1	7	9	3	2	4
1	2	3	4	5	6	7	8	9

```
REMOVE (A, i)
1  exchange A[i] ↔ A[HEAP-SIZE]
2  HEAP-SIZE ← HEAP-SIZE - 1
3  TOP-DOWN-MAX-HEAPIFY (A, i)
4  return A[HEAP-SIZE + 1]
```


Árvores binárias: *heaps*





operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*

vector A

16	8	10	4	7	9	3	2	1
1	2	3	4	5	6	7	8	9

```
REMOVE (A, i)
1  exchange A[i] ↔ A[HEAP-SIZE]
2  HEAP-SIZE ← HEAP-SIZE - 1
3  TOP-DOWN-MAX-HEAPIFY (A, i)
4  return A[HEAP-SIZE + 1]
```

Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort* ⊕

```
HEAPSORT (A)
1 BUILD-MAX-HEAP (A)
2 for i ← lenght(A) downto 2
3    do exchange A[1] ↔ A[i]
4       HEAP-SIZE ← HEAP-SIZE - 1
5       TOP-DOWN-MAX-HEAPIFY (A, 1)
```

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

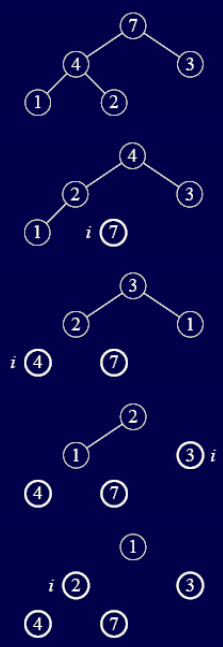
Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

método de ordenação *heapsort*




```
HEAPSORT (A)
1 BUILD-MAX-HEAP (A)
2 for i ← lenght(A) downto 2
3    do exchange A[1] ↔ A[i]
4       HEAP-SIZE ← HEAP-SIZE - 1
5       TOP-DOWN-MAX-HEAPIFY (A, 1)
```

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

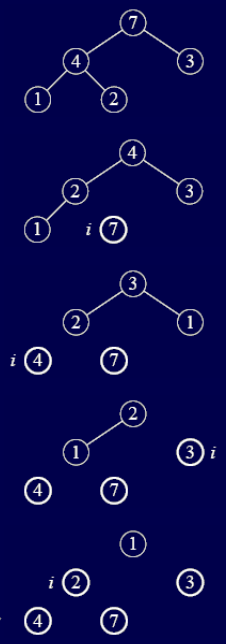
inserção e eliminação de nós

método de ordenação *heapsort*

vector final A

1	2	3	4	7
1	2	3	4	5

```
HEAPSORT (A)
1 BUILD-MAX-HEAP (A)
2 for i ← lenght(A) downto 2
3    do exchange A[1] ↔ A[i]
4       HEAP-SIZE ← HEAP-SIZE - 1
5       TOP-DOWN-MAX-HEAPIFY (A, 1)
```



(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

Árvores binárias: *heaps*

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

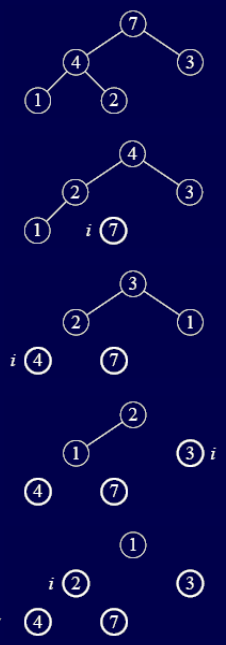
método de ordenação *heapsort*

Análise:
Build_Max_Heap demora O(n)
Cada uma das n-1 chamadas recursivas de Max-Heapify demora O(lg n)

T(n) = O(n) + n * O(lg n)

O(n lg n)

```
HEAPSORT (A)
1 BUILD-MAX-HEAP (A)
2 for i ← lenght(A) downto 2
3    do exchange A[1] ↔ A[i]
4       HEAP-SIZE ← HEAP-SIZE - 1
5       TOP-DOWN-MAX-HEAPIFY (A, 1)
```



(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

Árvores binárias : *heaps* e filas de espera por prioridades

Árvores binárias : *heaps e filas de espera por prioridades*

uma fila de espera por prioridades (*priority queue*) é uma estrutura de dados com duas operações básicas

inserção de elementos

saída do elemento com maior valor (prioritário)

Árvores binárias: *heaps* e filas de espera por prioridades

uma fila de espera por prioridades (*priority queue*) é uma estrutura de dados com duas operações básicas

inserção de elementos

saída do elemento com maior valor (prioritário)

tipo abstracto de dados *priority queue* :

```
void PQinit(int);
 int PQempty();
void PQinsert(Item);
Item PQdelmax();
```

(Robert Sedgewick. *Algorithms in C, 2nd edition.*)

Árvores binárias: *heaps* e filas de espera por prioridades

uma fila de espera por prioridades (*priority queue*) é uma estrutura de dados com duas operações básicas

inserção de elementos

saída do elemento com maior valor (prioritário)

metodologias de implementação :

Lista ordenada

Lista desordenada

Heap

Árvores binárias: *heaps e filas de espera por prioridades*

uma fila de espera por prioridades (*priority queue*) é uma estrutura de dados com duas operações básicas

inserção de elementos

saída do elemento com maior valor (prioritário)

Lista ordenada

```
PQinsert (A, X)
1  list_size ← list_size + 1
2  i ← 1
3  while PRIORITY(A[i]) < PRIORITY(X)
4     do i ← i + 1
5  for j ← list_size to i+1 inc -1
6     A[j] ← A[j-1]
7  A[i] ← X
```


Árvores binárias: *heaps e filas de espera por prioridades*

uma fila de espera por prioridades (*priority queue*) é uma estrutura de dados com duas operações básicas

inserção de elementos

saída do elemento com maior valor (prioritário)

Lista desordenada

```
PQinsert (A, X)
1  list_size ← list_size + 1
2  A[list_size] ← X
```

Árvores binárias: *heaps e filas de espera por prioridades*

uma fila de espera por prioridades (*priority queue*) é uma estrutura de dados com duas operações básicas

inserção de elementos

saída do elemento com maior valor (prioritário)

Heap

```
PQinsert (A, X)
1  INSERT_HEAP (A, X)
```


Árvores binárias: *heaps e filas de espera por prioridades*

uma fila de espera por prioridades (*priority queue*) é uma estrutura de dados com duas operações básicas

inserção de elementos

saída do elemento com maior valor (prioritário)

Heap

```
PQdelmax (A)
1  X ← REMOVE_HEAP (A, 1)
2  return X
```

Árvores binárias: *heaps e filas de espera por prioridades*

uma fila de espera por prioridades (*priority queue*) é uma estrutura de dados com duas operações básicas

inserção de elementos

saída do elemento com maior valor (prioritário)

metodologias de implementação :

Lista ordenada

Lista desordenada

Heap

Análise - Lista ordenada:

Construção da lista	$((n \times \lceil \log_2 n \rceil) - 2^{\lceil \log_2 n \rceil}) + 1 \rightarrow$ O(n log n)
Inserção	$(2 \times n) + 1 \rightarrow$ O(n)
Saída	O(1)

Árvores binárias: *heaps e filas de espera por prioridades*

uma fila de espera por prioridades (*priority queue*) é uma estrutura de dados com duas operações básicas

inserção de elementos

saída do elemento com maior valor (prioritário)

ordered array		
operation	expected analysis	
decrease key		O(n)
empty		O(1)
peek		O(1)
construct	$\left((n \times \lceil \log_2 n \rceil) - 2^{\lceil \log_2 n \rceil}\right) + 1$	O(n log n)
insert	$(2 \times n)+1$	O(n)
find	$\lceil \log_2 n \rceil$	O(log n)
remove		O(1)
size		O(1)

metodologias de implementação :

Lista ordenada

Lista desordenada

Heap

Análise - Lista ordenada:

Construção da lista	O(n log n)
Inserção	O(n)
Saída	O(1)

Árvores binárias: *heaps e filas de espera por prioridades*

uma fila de espera por prioridades (*priority queue*) é uma estrutura de dados com duas operações básicas

inserção de elementos

saída do elemento com maior valor (prioritário)

unordered array		
operation	expected analysis	
decrease key		O(n)
empty		O(1)
peek		O(n)
construct		O(n)
insert	2	O(1)
find		O(n)
remove		O(n)
size		O(1)

metodologias de implementação :

Lista ordenada

Lista desordenada

Heap

Análise - Lista desordenada:

Construção da lista	O(n)
Inserção	O(1)
Saída	O(n)

Árvores binárias: *heaps e filas de espera por prioridades*

uma fila de espera por prioridades (*priority queue*) é uma estrutura de dados com duas operações básicas

inserção de elementos

saída do elemento com maior valor (prioritário)

metodologias de implementação :

Lista ordenada

Lista desordenada

Heap

Análise - Heap:

Construção da lista	$(2 \times n) \rightarrow$ O(n)
Inserção	$[\log_2(n+1)] \times 2 \rightarrow$ O(log n)
Saída	$[\log_2 n] \times 2 \rightarrow$ O(log n)

Árvores binárias: *heaps e filas de espera por prioridades*

uma fila de espera por prioridades (*priority queue*) é uma estrutura de dados com duas operações básicas

inserção de elementos

saída do elemento com maior valor (prioritário)

binary heap		
operation	expected analysis	
decrease key	$\sum_{i=1}^{n} \lceil \log_2 i \rceil / n$	O(log n)
empty		O(1)
peek		O(1)
construct	$2 \times n$	O(n)
insert	$(\lceil \log_2 (n+1) \rceil) \times 2$	O(log n)
find		O(n)
remove	$(\lceil \log_2 n \rceil) \times 2$	O(log n)
size		O(1)

metodologias de implementação :

Lista ordenada

Lista desordenada

Heap

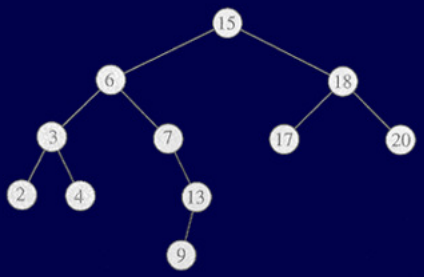
Análise - Heap:

Construção da lista	O(n)
Inserção	O(log n)
Saída	O(log n)

Árvores binárias : árvores binárias de pesquisa

Árvores binárias: árvores binárias de pesquisa

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)



são árvores binárias que obedecem à seguinte condição :

seja um nó ***x***

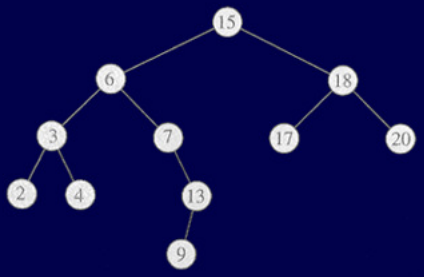
todos os nós da sub-árvore esquerda de ***x*** têm de ter valores menores ou iguais a ***x***

todos os nós da sub-árvore direita de ***x*** têm de ter valores maiores ou iguais a ***x***

utilizando o método *inorder* percorremos a árvore de modo a obter uma sequência ordenada – ascendente – dos dados dos nós
(< 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20 >)

Árvores binárias: árvores binárias de pesquisa

(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

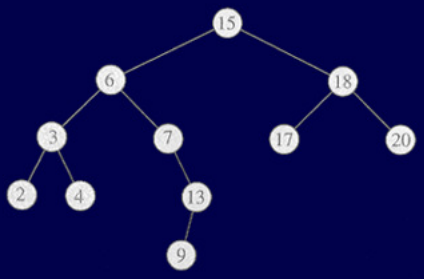


operações típicas :

inicialização e manutenção

inserção e eliminação de nós

Árvores binárias: árvores binárias de pesquisa



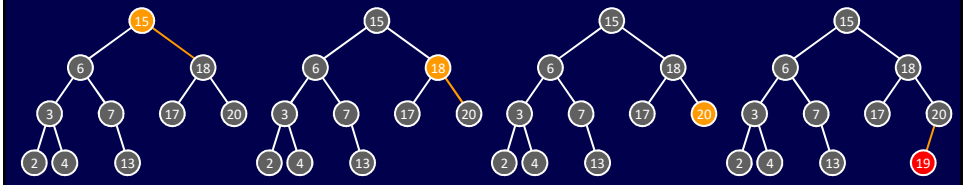
(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

operações típicas :

inicialização e manutenção

inserção e eliminação de nós (19)

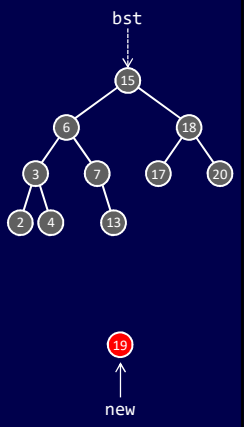
(método baseado na pesquisa em BSTs)



Árvores binárias: árvores binárias de pesquisa

inserção e eliminação de nós (exemplo de implementação em Linguagem C)

```
void insertNewNodeBST(INT_NODE *bst, INT_NODE *new)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //new é um apontador para o novo nó.
    //As ligações para a esq e para a dir de new valem NULL.

    INT_NODE *pAux=NULL, *pIter=bst;

    while (pIter != NULL) {
        pAux = pIter;
        if (DATA(new) < DATA(pIter)) pIter = LEFT(pIter);
        else pIter = RIGHT(pIter);
    }
    if (pAux == NULL) bst = new; //BST vazia
    else    if (DATA(new) < DATA(paux)) LEFT(pAux) = new;
            else RIGHT(pAux) = new;
    return();
}
```



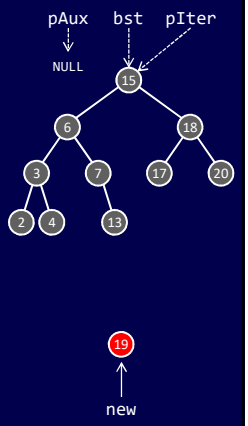
Exemplo:

Seja a seguinte BST e seja o valor a inserir, xNew = 19

Árvores binárias: árvores binárias de pesquisa

inserção e eliminação de nós (exemplo de implementação em Linguagem C)

```
void insertNewNodeBST(INT_NODE *bst, INT_NODE *new)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //new é um apontador para o novo nó.
    //As ligações para a esq e para a dir de new valem NULL.

    INT_NODE *pAux=NULL, *pIter=bst;

    while (pIter != NULL) {
        pAux = pIter;
        if (DATA(new) < DATA(pIter)) pIter = LEFT(pIter);
        else pIter = RIGHT(pIter);
    }
    if (pAux == NULL) bst = new; //BST vazia
    else    if (DATA(new) < DATA(paux)) LEFT(pAux) = new;
            else RIGHT(pAux) = new;
    return();
}
```



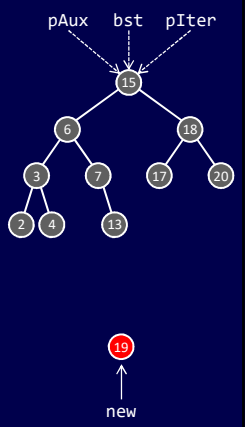
Exemplo:

Seja a seguinte BST e seja o valor a inserir, xNew = 19

Árvores binárias: árvores binárias de pesquisa

inserção e eliminação de nós (exemplo de implementação em Linguagem C)

```
void insertNewNodeBST(INT_NODE *bst, INT_NODE *new)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //new é um apontador para o novo nó.
    //As ligações para a esq e para a dir de new valem NULL.

    INT_NODE *pAux=NULL, *pIter=bst;

    while (pIter != NULL) {
        pAux = pIter;
        if (DATA(new) < DATA(pIter)) pIter = LEFT(pIter);
        else pIter = RIGHT(pIter);
    }
    if (pAux == NULL) bst = new; //BST vazia
    else    if (DATA(new) < DATA(paux)) LEFT(pAux) = new;
            else RIGHT(pAux) = new;
    return();
}
```



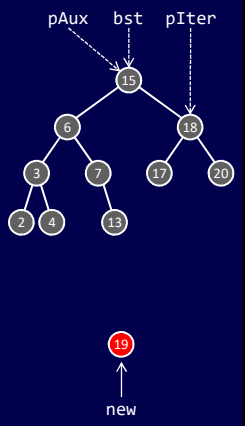
Exemplo:

Seja a seguinte BST e seja o valor a inserir, xNew = 19

Árvores binárias: árvores binárias de pesquisa

inserção e eliminação de nós (exemplo de implementação em Linguagem C)

```
void insertNewNodeBST(INT_NODE *bst, INT_NODE *new)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //new é um apontador para o novo nó.
    //As ligações para a esq e para a dir de new valem NULL.

    INT_NODE *pAux=NULL, *pIter=bst;

    while (pIter != NULL) {
        pAux = pIter;
        if (DATA(new) < DATA(pIter)) pIter = LEFT(pIter);
        else pIter = RIGHT(pIter);
    }
    if (pAux == NULL) bst = new; //BST vazia
    else   if (DATA(new) < DATA(paux)) LEFT(pAux) = new;
           else RIGHT(pAux) = new;
    return();
}
```



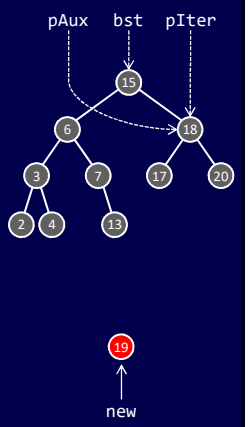
Exemplo:

Seja a seguinte BST e seja o valor a inserir, xNew = 19

Árvores binárias: árvores binárias de pesquisa

inserção e eliminação de nós (exemplo de implementação em Linguagem C)

```
void insertNewNodeBST(INT_NODE *bst, INT_NODE *new)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //new é um apontador para o novo nó.
    //As ligações para a esq e para a dir de new valem NULL.

    INT_NODE *pAux=NULL, *pIter=bst;

    while (pIter != NULL) {
        pAux = pIter;
        if (DATA(new) < DATA(pIter)) pIter = LEFT(pIter);
        else pIter = RIGHT(pIter);
    }
    if (pAux == NULL) bst = new; //BST vazia
    else   if (DATA(new) < DATA(paux)) LEFT(pAux) = new;
           else RIGHT(pAux) = new;
    return();
}
```



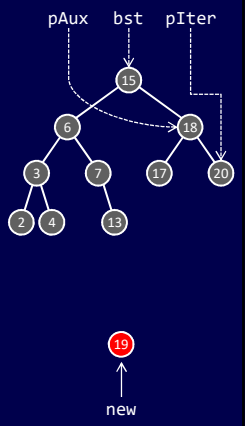
Exemplo:

Seja a seguinte BST e seja o valor a inserir, xNew = 19

Árvores binárias: árvores binárias de pesquisa

inserção e eliminação de nós (exemplo de implementação em Linguagem C)

```
void insertNewNodeBST(INT_NODE *bst, INT_NODE *new)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //new é um apontador para o novo nó.
    //As ligações para a esq e para a dir de new valem NULL.

    INT_NODE *pAux=NULL, *pIter=bst;

    while (pIter != NULL) {
        pAux = pIter;
        if (DATA(new) < DATA(pIter)) pIter = LEFT(pIter);
        else pIter = RIGHT(pIter);
    }
    if (pAux == NULL) bst = new; //BST vazia
    else   if (DATA(new) < DATA(paux)) LEFT(pAux) = new;
           else RIGHT(pAux) = new;
    return();
}
```



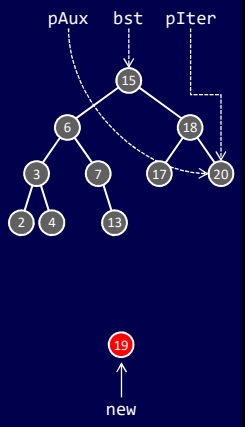
Exemplo:

Seja a seguinte BST e seja o valor a inserir, xNew = 19

Árvores binárias: árvores binárias de pesquisa

inserção e eliminação de nós (exemplo de implementação em Linguagem C)

```
void insertNewNodeBST(INT_NODE *bst, INT_NODE *new)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //new é um apontador para o novo nó.
    //As ligações para a esq e para a dir de new valem NULL.

    INT_NODE *pAux=NULL, *pIter=bst;

    while (pIter != NULL) {
        pAux = pIter;
        if (DATA(new) < DATA(pIter)) pIter = LEFT(pIter);
        else pIter = RIGHT(pIter);
    }
    if (pAux == NULL) bst = new; //BST vazia
    else   if (DATA(new) < DATA(paux)) LEFT(pAux) = new;
           else RIGHT(pAux) = new;
    return();
}
```



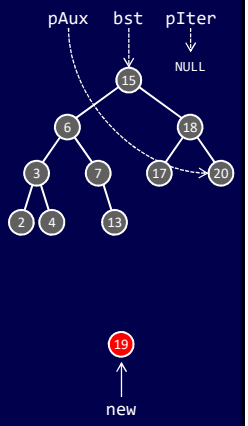
Exemplo:

Seja a seguinte BST e seja o valor a inserir, xNew = 19

Árvores binárias: árvores binárias de pesquisa

inserção e eliminação de nós (exemplo de implementação em Linguagem C)

```
void insertNewNodeBST(INT_NODE *bst, INT_NODE *new)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //new é um apontador para o novo nó.
    //As ligações para a esq e para a dir de new valem NULL.

    INT_NODE *pAux=NULL, *pIter=bst;

    while (pIter != NULL) {
        pAux = pIter;
        if (DATA(new) < DATA(pIter)) pIter = LEFT(pIter);
        else pIter = RIGHT(pIter);
    }
    if (pAux == NULL) bst = new; //BST vazia
    else   if (DATA(new) < DATA(paux)) LEFT(pAux) = new;
           else RIGHT(pAux) = new;
    return();
}
```



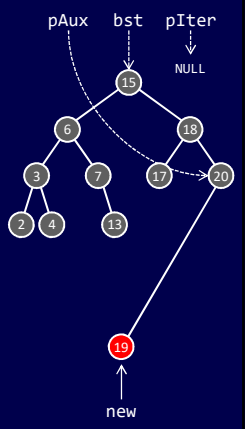
Exemplo:

Seja a seguinte BST e seja o valor a inserir, xNew = 19

Árvores binárias: árvores binárias de pesquisa

inserção e eliminação de nós (exemplo de implementação em Linguagem C)

```
void insertNewNodeBST(INT_NODE *bst, INT_NODE *new)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //new é um apontador para o novo nó.
    //As ligações para a esq e para a dir de new valem NULL.

    INT_NODE *pAux=NULL, *pIter=bst;

    while (pIter != NULL) {
        pAux = pIter;
        if (DATA(new) < DATA(pIter)) pIter = LEFT(pIter);
        else pIter = RIGHT(pIter);
    }
    if (pAux == NULL) bst = new; //BST vazia
    else   if (DATA(new) < DATA(paux)) LEFT(pAux) = new;
           else RIGHT(pAux) = new;
    return();
}
```



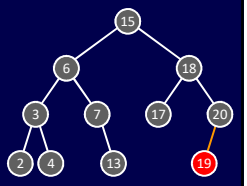
Exemplo:

Seja a seguinte BST e seja o valor a inserir, xNew = 19

Árvores binárias: árvores binárias de pesquisa

inserção e eliminação de nós (exemplo de implementação em Linguagem C)

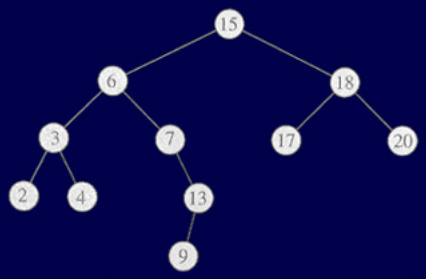
```
void insertNewNodeBST(INT_NODE *bst, INT_NODE *new)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //new é um apontador para o novo nó.
    //As ligações para a esq e para a dir de new valem NULL.

    INT_NODE *pAux=NULL, *pIter=bst;

    while (pIter != NULL) {
        pAux = pIter;
        if (DATA(new) < DATA(pIter)) pIter = LEFT(pIter);
        else pIter = RIGHT(pIter);
    }
    if (pAux == NULL) bst = new; //BST vazia
    else   if (DATA(new) < DATA(paux)) LEFT(pAux) = new;
           else RIGHT(pAux) = new;
    return();
}
```



Exemplo:

Seja a seguinte BST e seja o valor a inserir, xNew = 19

Árvores binárias: árvores binárias de pesquisa



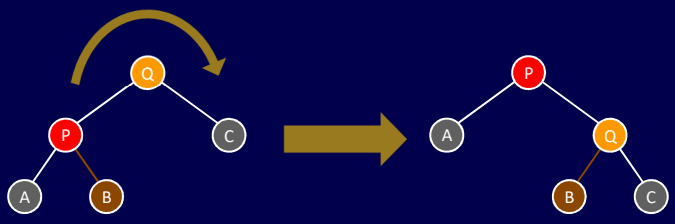
(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

operações típicas :

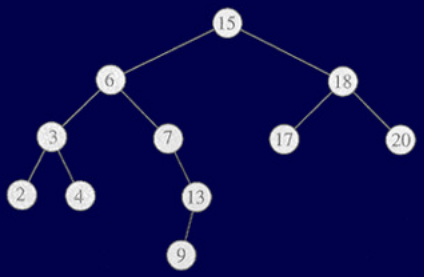
inicialização e manutenção

inserção e eliminação de nós

(rotação à direita)



Árvores binárias: árvores binárias de pesquisa

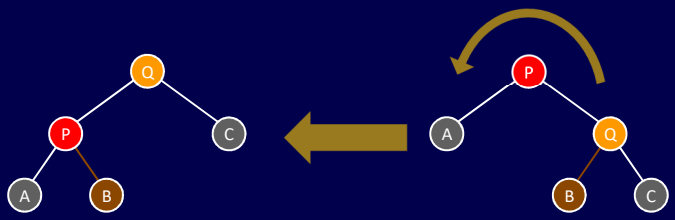


(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

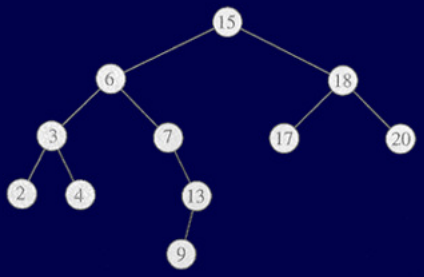
operações típicas :

inicialização e manutenção

inserção e eliminação de nós
(rotação à esquerda)



Árvores binárias: árvores binárias de pesquisa

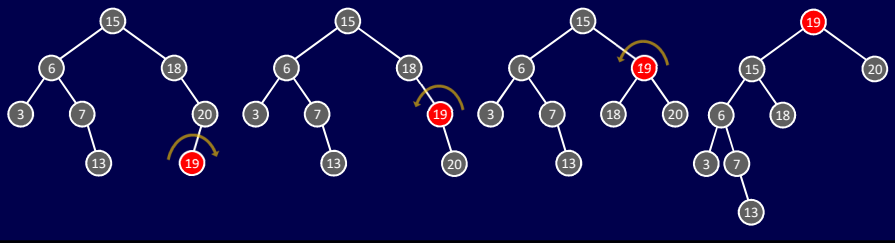


(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

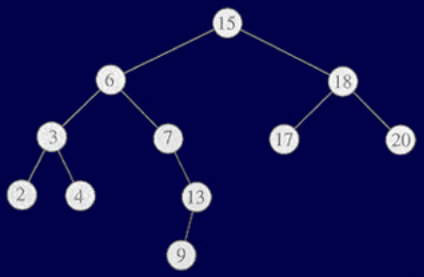
operações típicas :

inicialização e manutenção

inserção e eliminação de nós 19
(inserção na raiz)



Árvores binárias: árvores binárias de pesquisa



(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

operações típicas :

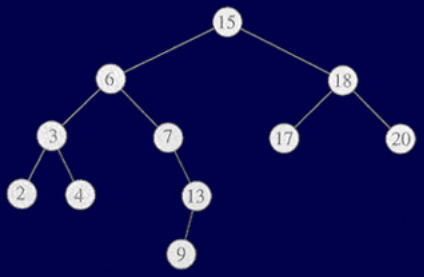
inicialização e manutenção

inserção e **eliminação** de nós

Existem três casos a considerar:

1. Eliminar um nó sem filhos (folha)
   Basta eliminar o nó da árvore.
2. Eliminar um nó com um filho
   Substitui-se esse nó pelo seu filho (mantendo este as suas sub-árvores).
3. Eliminar um nó com dois filhos
   Substitui-se pelo seu antecessor ou sucessor R (ao percorrer a árvore utilizando o método *in-order*), eliminando este da sua posição anterior.

Árvores binárias: árvores binárias de pesquisa



(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

operações típicas :

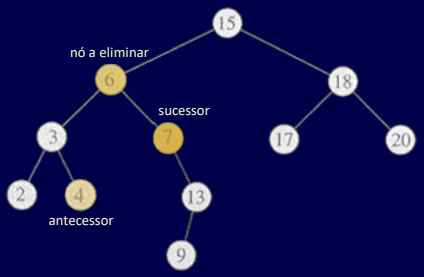
inicialização e manutenção

inserção e **eliminação** de nós

Existem três casos a considerar:

1. Eliminar um nó sem filhos (folha)
   Basta eliminar o nó da árvore.
2. Eliminar um nó com um filho
   Substitui-se esse nó pelo seu filho (mantendo este as suas sub-árvores).
3. Eliminar um nó com dois filhos
   Substitui-se pelo seu antecessor ou sucessor R (ao percorrer a árvore utilizando o método *in-order*), eliminando este da sua posição anterior.
   (relembra-se que, neste caso, o seu antecessor é o nó mais à direita da sua sub-árvore esquerda e o seu sucessor é o nó mais à esquerda da sua sub-árvore direita)

Árvores binárias: árvores binárias de pesquisa



(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

operações típicas :

inicialização e manutenção

inserção e **eliminação** de nós

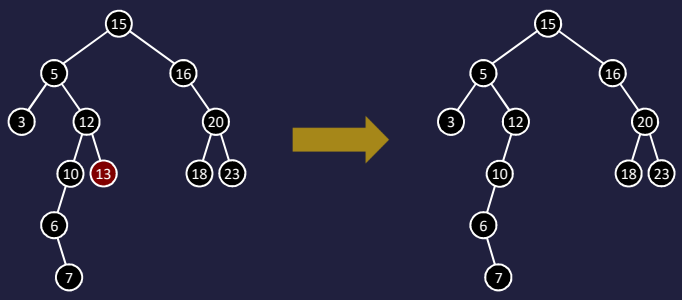
Existem três casos a considerar:

1. Eliminar um nó sem filhos (folha)
   Basta eliminar o nó da árvore.
2. Eliminar um nó com um filho
   Substitui-se esse nó pelo seu filho (mantendo este as suas sub-árvores).
3. Eliminar um nó com dois filhos
   Substitui-se pelo seu antecessor ou sucessor R (ao percorrer a árvore utilizando o método *in-order*), eliminando este da sua posição anterior.
   (relembra-se que, neste caso, o seu antecessor é o nó mais à direita da sua sub-árvore esquerda e o seu sucessor é o nó mais à esquerda da sua sub-árvore direita)

Árvores binárias: árvores binárias de pesquisa

inserção e eliminação de nós

1. Eliminar um nó sem filhos (folha)
   Basta eliminar esse nó da árvore.

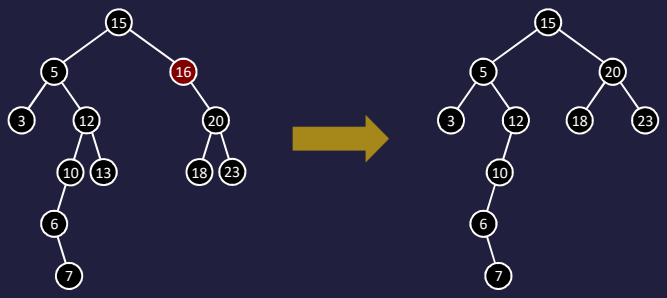


Árvores binárias: árvores binárias de pesquisa

inserção e eliminação de nós

2. Eliminar um nó com um filho

   Substitui-se esse nó pelo seu filho (mantendo este as suas sub-árvores).

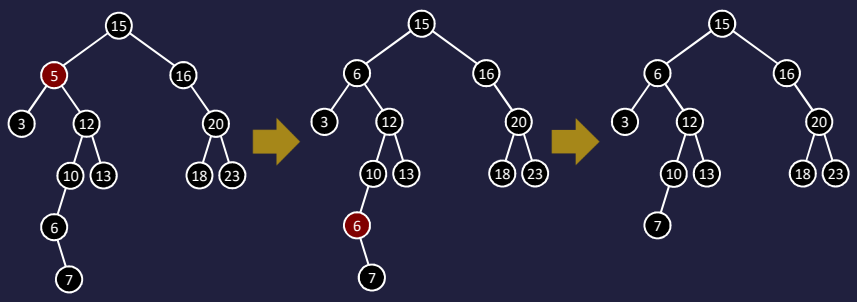


Árvores binárias: árvores binárias de pesquisa

inserção e eliminação de nós

3. Eliminar um nó com dois filhos

   Substitui-se pelo seu antecessor ou sucessor R (ao percorrer a árvore utilizando o método *in-order*), eliminando este da sua posição anterior.



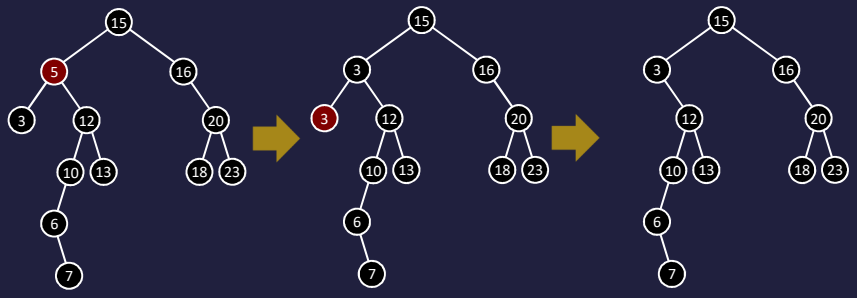
Percurso in-order: 3, 5, 6, 7, 10, 12, 13, ...

Árvores binárias: árvores binárias de pesquisa

inserção e eliminação de nós

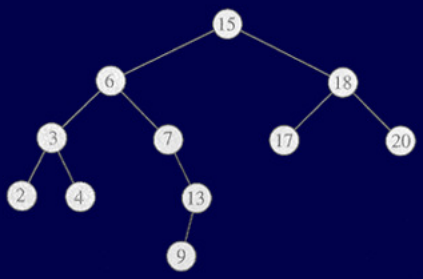
3. Eliminar um nó com dois filhos (2ª hipótese)

   Substitui-se pelo seu antecessor ou sucessor R (ao percorrer a árvore utilizando o método *in-order*), eliminando este da sua posição anterior.



Percurso in-order: 3, 5, 6, 7, 10, 12, 13, …

Árvores binárias: árvores binárias de pesquisa



(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

operações típicas :

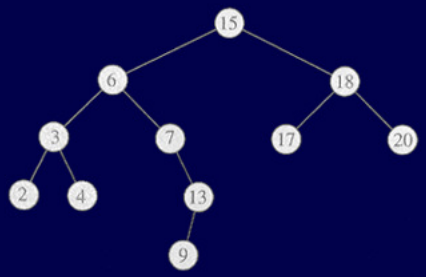
inicialização e manutenção

inserção e **eliminação** de nós

Existem três casos a considerar:

1. Eliminar um nó sem filhos (folha)
   Basta eliminar o nó da árvore.
2. Eliminar um nó com um filho
   Substitui-se esse nó pelo seu filho (mantendo este as suas sub-árvores).
3. Eliminar um nó com dois filhos
   A utlilização de um só tipo de substituto (antecessor ou sucessor) conduz a BSTs desequilibradas, pelo que um bom algoritmo de eliminação de nós com dois filhos em BSTs deve alternar esta escolha.

Árvores binárias: árvores binárias de pesquisa



(T. Cormen et al. *Introduction to Algorithms, 2nd edition.*)

operações típicas :

inicialização e manutenção

inserção e eliminação de nós

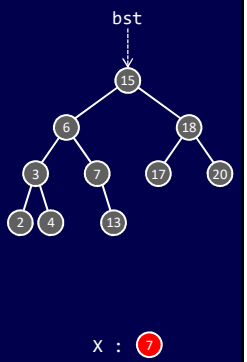
pesquisa de um elemento

```
TREE-SEARCH(x, k)
1  if x = NIL or k = key[x]
2     then return x
3  if k < key[x]
4     then return TREE-SEARCH(left[x], k)
5     else  return TREE-SEARCH(right[x], k)
```


Árvores binárias: árvores binárias de pesquisa

pesquisa de um elemento (exemplo de implementação em Linguagem C)

```
INT_NODE *pesquisaBinariaIterativa(INT_NODE *bst, int x)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //x é o valor inteiro que se pretende encontrar.
    //O resultado é um apontador para o nó que contém o valor x.
    //Ou NULL caso a pesquisa não seja bem sucedida.

    while (bst != NULL && DATA(bst) != x)
    {
        if (DATA(bst) > x)
            bst=LEFT(bst);
        else
            bst=RIGHT(bst);
    }

    return (bst);
}
```



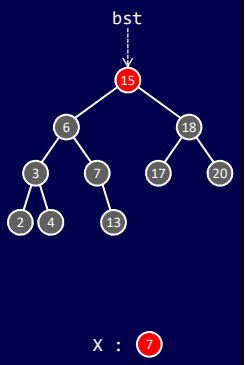
Exemplo:

Seja a seguinte BST e seja o valor a procurar, x = 7

Árvores binárias: árvores binárias de pesquisa

pesquisa de um elemento (exemplo de implementação em Linguagem C)

```
INT_NODE *pesquisaBinariaIterativa(INT_NODE *bst, int x)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //x é o valor inteiro que se pretende encontrar.
    //O resultado é um apontador para o nó que contém o valor x.
    //Ou NULL caso a pesquisa não seja bem sucedida.

    while (bst != NULL && DATA(bst) != x)
    {
        if (DATA(bst) > x)
            bst=LEFT(bst);
        else
            bst=RIGHT(bst);
    }

    return (bst);
}
```



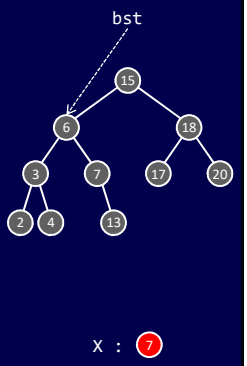
Exemplo:

Seja a seguinte BST e seja o valor a procurar, x = 7

Árvores binárias: árvores binárias de pesquisa

pesquisa de um elemento (exemplo de implementação em Linguagem C)

```
INT_NODE *pesquisaBinariaIterativa(INT_NODE *bst, int x)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //x é o valor inteiro que se pretende encontrar.
    //O resultado é um apontador para o nó que contém o valor x.
    //Ou NULL caso a pesquisa não seja bem sucedida.

    while (bst != NULL && DATA(bst) != x)
    {
        if (DATA(bst) > x)
            bst=LEFT(bst);
        else
            bst=RIGHT(bst);
    }

    return (bst);
}
```



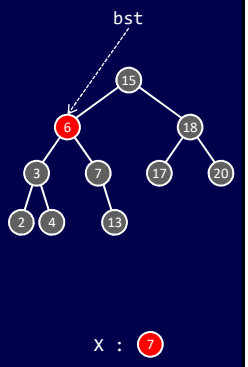
Exemplo:

Seja a seguinte BST e seja o valor a procurar, x = 7

Árvores binárias: árvores binárias de pesquisa

pesquisa de um elemento (exemplo de implementação em Linguagem C)

```
INT_NODE *pesquisaBinariaIterativa(INT_NODE *bst, int x)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //x é o valor inteiro que se pretende encontrar.
    //O resultado é um apontador para o nó que contém o valor x.
    //Ou NULL caso a pesquisa não seja bem sucedida.

    while (bst != NULL && DATA(bst) != x)
    {
        if (DATA(bst) > x)
            bst=LEFT(bst);
        else
            bst=RIGHT(bst);
    }

    return (bst);
}
```



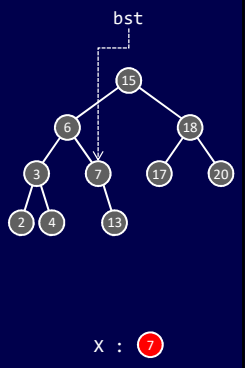
Exemplo:

Seja a seguinte BST e seja o valor a procurar, x = 7

Árvores binárias: árvores binárias de pesquisa

pesquisa de um elemento (exemplo de implementação em Linguagem C)

```
INT_NODE *pesquisaBinariaIterativa(INT_NODE *bst, int x)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //x é o valor inteiro que se pretende encontrar.
    //O resultado é um apontador para o nó que contém o valor x.
    //Ou NULL caso a pesquisa não seja bem sucedida.

    while (bst != NULL && DATA(bst) != x)
    {
        if (DATA(bst) > x)
            bst=LEFT(bst);
        else
            bst=RIGHT(bst);
    }

    return (bst);
}
```



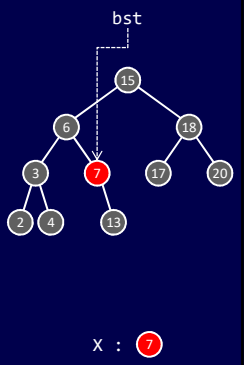
Exemplo:

Seja a seguinte BST e seja o valor a procurar, x = 7

Árvores binárias: árvores binárias de pesquisa

pesquisa de um elemento (exemplo de implementação em Linguagem C)

```
INT_NODE *pesquisaBinariaIterativa(INT_NODE *bst, int x)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //x é o valor inteiro que se pretende encontrar.
    //O resultado é um apontador para o nó que contém o valor x.
    //Ou NULL caso a pesquisa não seja bem sucedida.

    while (bst != NULL && DATA(bst) != x)
    {
        if (DATA(bst) > x)
            bst=LEFT(bst);
        else
            bst=RIGHT(bst);
    }

    return (bst);
}
```



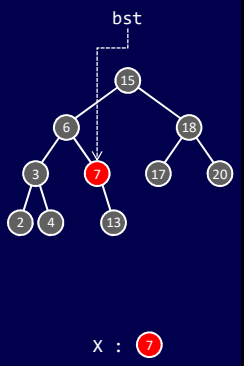
Exemplo:

Seja a seguinte BST e seja o valor a procurar, x = 7

Árvores binárias: árvores binárias de pesquisa

pesquisa de um elemento (exemplo de implementação em Linguagem C)

```
INT_NODE *pesquisaBinariaIterativa(INT_NODE *bst, int x)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //x é o valor inteiro que se pretende encontrar.
    //O resultado é um apontador para o nó que contém o valor x.
    //Ou NULL caso a pesquisa não seja bem sucedida.

    while (bst != NULL && DATA(bst) != x)
    {
        if (DATA(bst) > x)
            bst=LEFT(bst);
        else
            bst=RIGHT(bst);
    }

    return (bst);
}
```



Exemplo:

Seja a seguinte BST e seja o valor a procurar, x = 7

Árvores binárias: árvores binárias de pesquisa

pesquisa de um elemento (exemplo de implementação recursiva em Linguagem C)

```
INT_NODE *pesquisaBinaria(INT_NODE *bst, int x)
{
    //bst é um apontador para a árvore binária de pesquisa.
    //x é o valor inteiro que se pretende encontrar.
    //O resultado é um apontador para o nó que contém o valor x.
    //Ou NULL caso a pesquisa não seja bem sucedida.

    if (bst == NULL || DATA(bst) == x)  return (bst);

    if (DATA(bst) > x)
        return pesquisaBinaria(LEFT(bst), x);
    else
        return pesquisaBinaria(RIGHT(bst), x);
}
```

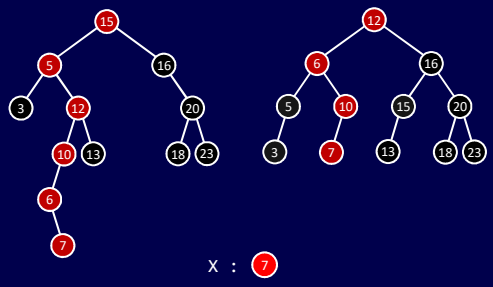

Árvores binárias: árvores binárias de pesquisa

operações típicas :

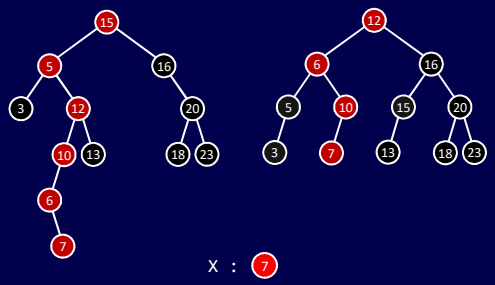
inicialização e manutenção

inserção e eliminação de nós

pesquisa de um elemento



Árvores binárias: árvores binárias de pesquisa



operações típicas :

inicialização e manutenção

inserção e eliminação de nós

pesquisa de um elemento

a eficácia das BSTs como suporte de algoritmos de pesquisa depende do equilíbrio da referida BST

Árvores binárias: árvores binárias de pesquisa

Estratégias de equilíbrio de árvores binárias

1. Não equilibrar
   - Podemos acabar por ter uma árvore com uma profundidade elevada (deteriorando assim o desempenho como BST)
2. Equilíbrio rigoroso
   - A árvore deve permanecer sempre perfeitamente equilibrada (pode ser exigente em termos de desempenho)
3. Um bom equilíbrio
   - Admite-se que a árvore possa apresentar algum desequilíbrio (este relaxamento na exigência do equilíbrio pode significar um ganho no desempenho)
4. Adaptar no acesso
   - Auto-adaptável
     (esta adaptação, geralmente associada à ideia de amortização, tem como intuito melhorar a eficácia das operações seguintes)