8. Árvores

Fernando Silva

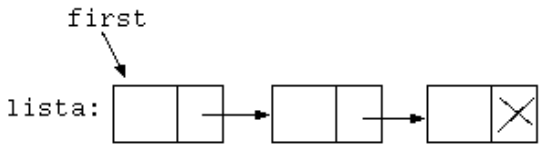
DCC-FCUP

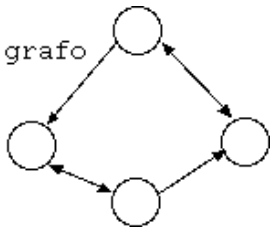
Estruturas de Dados

Árvores - estruturas não lineares (1)

- Uma **lista** é um exemplo de uma estrutura de dados *linear*, pois cada elemento tem:
  - um predecessor único, excepto o primeiro elemento da lista;
  - tem um sucessor único, excepto o último elemento da lista.

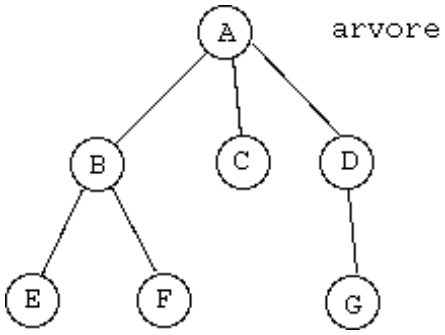
  As pilhas e filas são outros exemplos.
  Existem outros tipos de estruturas?
- Um **grafo** é uma estrutura de dados *não-linear*, pois os seus elementos, designados por nós, podem ter mais de um predecessor ou mais de um sucessor.





Árvores - estruturas não lineares (2)

- As **árvores** são um caso especial de grafos, em que que cada elemento (nó) tem zero ou mais sucessores, mas tem apenas um predecessor, excepto o primeiro nó, a raiz da árvore.



- São estruturas naturalmente adequadas para representar informação organizada em hierarquias.
  - Um exemplo comum é a estrutura de directórios (ou pastas) de umsistema de ficheiros.

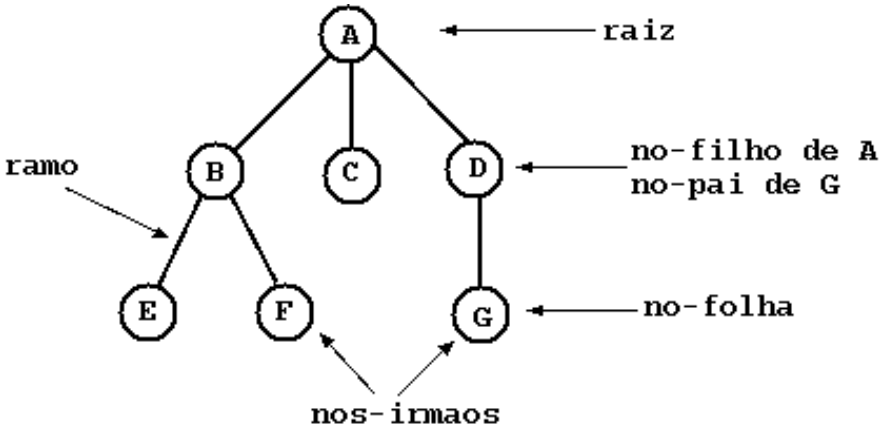
Árvores - definição

- Uma árvore é um grafo constituído por um conjunto de nós e um conjunto de arcos que ligam pares de nós, em que:
  - cada arco liga um nó-pai a um ou mais nós-filho;
  - todos os nós, com excepção da raíz, têm um nó-pai.
- Definição recursiva:

Uma árvore T de aridade n é constituída por um conjunto finito de nós, tal que:

- ou o conjunto é vazio, $T = \emptyset$; ou
- consiste de uma raíz r e de $n \geq 0$ sub-árvores distintas, $T = \{r, T_0, T_1, \ldots, T_{n-1}\}$.
  - ★ as sub-árvores T_i são árvores cujos nós raíz r_i são nós-filho de r.

Árvores - conceitos/terminologia



- ao predecessor (único) de um nó, chama-se **nó-pai**
- os seus sucessores são os **nós-filho**
- o **grau** de um nó é o número sub-árvores (ou nós-filho) que descendem desse nó.
- um nó-folha não tem filhos, tem grau 0.
- um nó-raiz não tem pai
- os arcos que ligam os nós, chamam-se **ramos**

Árvores - terminologia

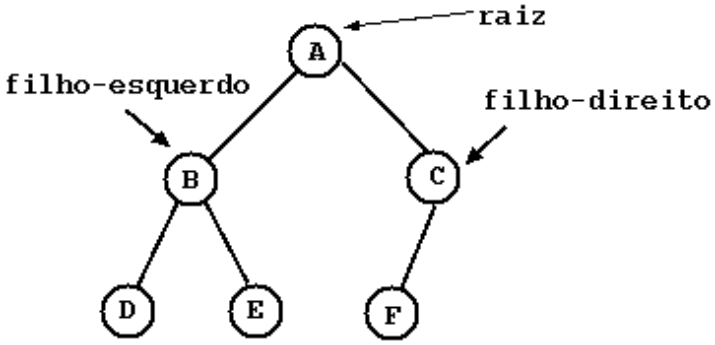
- chama-se **caminho** a uma sequência de ramos entre dois nós
  - Uma propriedade importante de uma árvore é que existe um e apenas um caminho entre dois quaisquer nós de uma árvore.
- o comprimento de um caminho é o número de ramos nele contido
- a profundidade de um nó **n** é o comprimento do caminho de n até à raíz; a profundidade da raíz é zero.
- a altura de um nó é o comprimento do caminho desde esse nó até ao seu nó-folha mais profundo (a altura de um nó folha é zero).
- a altura de uma árvore é a altura da raíz (i.e. o comprimento do maior caminho de um nó-folha até à raíz).

Árvores Binárias - definição informal

Uma **árvore-binária** é constituída por um conjunto finito de nós. Se o conjunto for vazio, a árvore diz-se **vazia**, caso contrário obdece às seguintes regras:

1. possui um nó especial, a **raiz** da árvore.
2. cada nó possui no máximo dois filhos, **filho-esquerdo** e **filho-direito**.
3. cada nó, excepto a raíz, possui exactamente um **nó-pai**.

ou dito de forma mais simples,
Árvores binárias são árvores em que cada nó tem 0, 1 ou 2 filhos.



Árvores Binárias - conceitos

- Uma **árvore binária totalmente preenchida** é uma árvore binária em que todos os nós, excepto os nós-folha, têm 2 filhos.
- Uma **árvore binária perfeita** é uma árvore binária totalmente preenchida em que todos os nós-folha estão à mesma profundidade.
- Uma **Árvore binária completa** é uma árvore-binária em que todos os níveis, excepto possivelmente o último, estão completamente preenchidos e todos os nós estão o mais à esquerda possível.

- A profundidade de uma árvore binária é determinada pelo maior nível de qualquer nó folha.

Profundidade de Árvores Binárias (1)

- Como determinar o número de nós de uma árvore binária perfeita de profundidade d?
  - O número de nós total é a soma dos nós dos níveis de 0 a d, i.e.

    nível 0 – $2^0 = 1$ nó
    nível 1 – $2^1 = 2$ nós
    ... ...
    nível d – 2^d nós
    Soma – $2^0 + 2^1 + 2^2 + \ldots + 2^d = \sum_{j=0}^{d} 2^j$

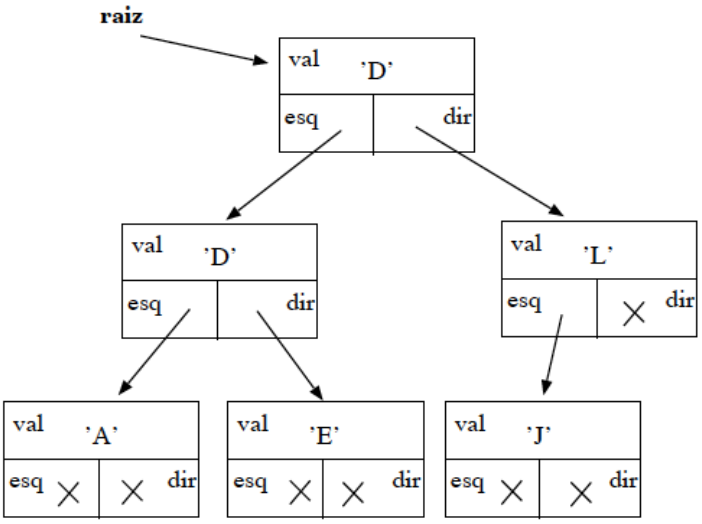
    Por indução, Soma$= 2^{d+1} - 1$.

Profundidade de Árvores Binárias (2)

- Conhecido o número de nós de uma árvore binária perfeita, é possível determinar a sua profundidade d:
  - $n = 2^{d+1} - 1 \implies d = log_2(n+1) - 1$
  - Exemplos:
    - $10 = log_2(1024)$
    - $20 = log_2(1000000)$
- Portanto, apesar de uma árvore binária poder conter muitos nós, a distância da raiz a qualquer folha é relativamente pequena.
- Isto é excelente pois significa que os algoritmos sobre árvores (inserir, remover, procurar), requerem apenas percorrer um caminho cuja profundidade é logarítmica no número de nós da árvore determinando a complexidade do algoritmo.

Árvore binárias em Java (1)

- A caracterização de um nó é determinante.
- Vejamos primeiro uma figura a ilustrar o encadeamento de nós de uma árvore de caracteres.



Árvores Binárias em Java (2) – class BTNode

A árvore fica definida pelo encadeamento de nós do tipo `BTNode`.
Definição para valores inteiros:

```
class BTNode {
    int    val; // valor do nó
    BTNode esq; // referência para o filho esquerdo
    BTNode dir; // referência para o filho direito

    // construtor de novo nó
    BTNode(int v, BTNode e, BTNode d) {
        val= v; // valor v, e referencias para
        esq= e; // filho esquerdo e
        dir= d; // filho direito
    }
}
```

Árvores Binárias em Java (2) – class BTNode<E>

Vejamos a definição genérica para um nó de uma árvore:

```
class BTNode<E extends Comparable<E>> {
    E         val; // valor do nó
    BTNode<E> esq; // filho esquerdo
    BTNode<E> dir; // filho direito

    // construtor de novo nó
    BTNode(E v, BTNode<E> e, BTNode<E> d) {
        val= v;
        esq= e;
        dir= d;
    }
    // métodos ...
}
```

Com a definição da classe `BTNode` podemos organizar todas os métodos sobre árvores.
A comparação de objectos do tipo `E` requer que esteja implementado o método `compareTo()` na classe que define `E`.

Árvores Binárias em Java (3) – métodos

Alguns métodos:

```
// verifica se um dado nó (que invoca o método) é folha
public boolean folha() {
    return (esq==null) && (dir==null);
}
// Determina o num. nós da árvore cuja raíz é t
public int size(BTNode<E> t) {
    if (t==null)
        return 0;
    else
        return 1 + size(t.esq) + size(t.dir);
}
// Determina a profundidade da +arvore com raíz t
public int prof(BTNode<E> t) {
    if (t==null)
        return -1;
    else
        return 1 + Math.max(prof(t.esq), prof(t.dir));
}
```

Árvores Binárias em Java (4) – classe BTree

Representação mais completa de árvore com duas classes, uma caracteriza um nó e outra define a árvore (protege a raíz).

```
class BTNode<E extends Comparable<E>> {
    E         val; // valor do nó
    BTNode<E> esq; // filho esquerdo
    BTNode<E> dir; // filho direito

    // construtor de novo nó
    BTNode(E v, BTNode<E> e, BTNode<E> d) {
        val= v; esq= e; dir= d;
    }
}
// Árvore + métodos
class BTree<E> {
    private BTNode<E> root; // raiz da árvore

    BTree() {root= null;} // cria árvore vazia
    // metodos
    public boolean isEmpty() { return root==null;}
}
```


Árvores Binárias em Java (5) – classe BTree

Representação de árvore com a definição do nó interna à classe árvore.

```
class BTree<E extends Comparable<E>> {
    // nó da árvore -- classe privada
    class BTNode {
        E         val; // valor do nó
        BTNode<E> esq; // filho esquerdo
        BTNode<E> dir; // filho direito

        // construtor de novo nó
        BTNode(E v, BTNode e, BTNode d) {
            val= v; esq= e; dir= d;
        }
    }
    private BTNode root; // raiz da árvore

    BTree() { root= null;} // cria árvore vazia

    // metodos
    public boolean isEmpty() { return root==null;}
}
```

Árvores Binárias em Java (5) – classe BTree

Vantagens de usar duas classes:

- definição mais clara de árvore vazia e raíz da árvore,
- atributo raíz fica mais protegido,
- atributos do `BTNode` não ficam acessíveis a outras classes.

Template os métodos da classe `BTree`, com a classe `BTNode` interna:

```
class BTree<E extends Comparable<E>> {
    class BTNode { ... }
    private BTNode root; // raiz da árvore

    // ...
    public TipoDados nomeMetodo(parametros) {
        nomeMetodo(root, parametros);
    }
    private TipoDados nomeMetodo(BTNode t, parametros) {
        //...
    }
}
```


Árvores Binárias de Pesquisa – definição

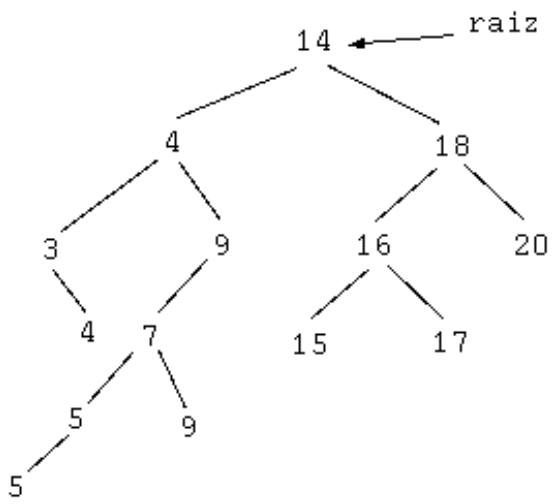
Uma árvore binária T diz-se de pesquisa se:

- T for vazia, ou
- cada nó de T contém uma chave que satisfaz as condições seguintes:
  - todas as chaves (se existirem) na sub-árvore esquerda da raíz precedem a chave da raíz,
  - a chave da raíz precede as chaves (se existirem) na sub-árvore direita,
  - as sub-árvores esquerda e direita da raíz também são árvores de pesquisa.

Árvores Binárias de Pesquisa - exemplo construção

Munidos da definição, podemos agora construir uma árvore binária de pesquisa:

- o primeiro valor fica na raíz da árvore.
- os seguintes são à esquerda ou direita da raíz, obedecendo à relação de ordem, como folhas e em níveis cada vez mais baixos.

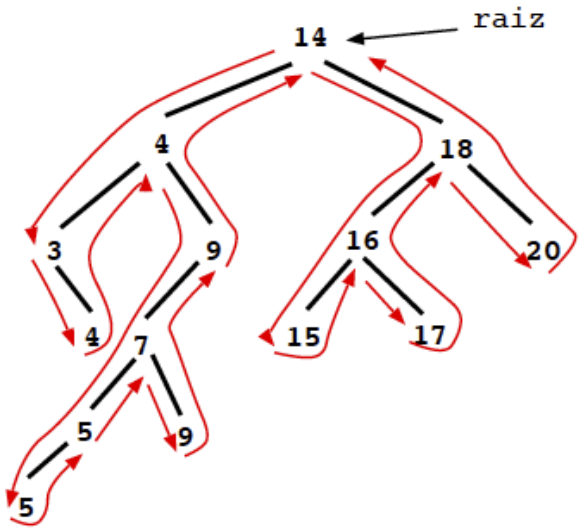


Sequência de valores dada:
14,18,4,9,7,16,3,5,4,17,20,15,9,5

Árvores Binárias de Pesquisa - exemplo visita

Interessante notar que:

- Percorrendo esta árvore com determinado critério, permite-nos obter a mesma sequência ordenada por ordem crescente.
- Trata-se de uma visita em profundidade (ou depth-first).



visita inorder:
3,4,4,5,5,7,9,9,
14,15,16,17,18,20

Árvores Binárias de Pesquisa (1) - insertBTNode()

A construção de uma árvore faz-se, invocando o método `insertBTNode()` para cada novo valor a inserir. Consideramos o valor do nó como `int`.

```
public void insertBTNode(int k) {
    root= insertBTNode(root, k);
}
private BTNode insertBTNode(BTNode t, int k) {
    //inserir na árvore com raíz t o valor k
    //se árvore vazia, novo nó com k fica a raíz
    if(t==null)
        return new BTNode(k,null,null);
    else { // senão tenta numa das sub-árvores
        if (k <= t.val)
            t.esq= insertBTNode(t.esq, k);
        else
            t.dir= insertBTNode(t.dir, k);
    }
    return t;
}
```

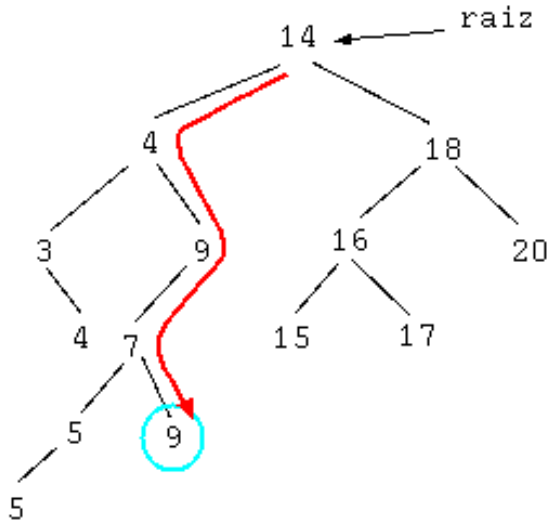

Árvores Binárias de Pesquisa (2) - insertBTNode()

Método `insertBTNode()` em que o valor de `BTNode` é um objecto genérico que implementa `Comparable`.

```
public void insertBTNode(E k) {
    root= insertBTNode(root, k);
}
private BTNode insertBTNode(BTNode t, E k) {
    //inserir na árvore com raíz t o valor k
    //se árvore vazia, novo nó com k fica a raíz
    if(t==null)
        return new BTNode(k,null,null);
    else { // senão tenta numa das sub-árvores
        if (k.compareTo(t.val) <= 0)
            t.esq= insertBTNode(t.esq, k);
        else
            t.dir= insertBTNode(t.dir, k);
    }
    return t;
}
```

Árvores Binárias de Pesquisa - exemplo inserir

Ilustração do percurso na árvore quando se pretende inserir um dado valor.



seq: 14,18,4,9,7,16,3,5,4
17,20,15,9,5

Árvores Binárias de Pesquisa - BuildTree

Classe principal lê do standard-input uma sequência de inteiros e constrói a árvore e depois imprima os valores "inorder".

```
class BuildTree {
    public static void main(String args[]) {
        Scanner scan= new Scanner(System.in);
        BTree t= new BTree(); // vazia
        int nnos, k;

        nnos= scan.nextInt();//num. valores na sequência
        for (int i= 0; i<nnos; i++) {
            k= scan.nextInt();
            t.insertBTNode(k);
        }
        System.out.println("Sequencia ordenada: ");
        t.printInorder();
    }
}
```

Árvores binárias – visita em profundidade

As regras de construção de uma árvore binária de pesquisa garantem que a informação em todos os nós da sub-árvore esquerda é menor do que a raiz que por sua vez é menor ou igual do que qualquer dos valores na sub-árvore direita.

O método de pesquisa **inorder** é particularmente útil para pesquisar os nós de uma árvore de pesquisa binária em que a ordem dos valores (chaves de pesquisa) é crescente. Assim como permite listar por ordem a sequência de valores da árvore.

O método de pesquisa **inorder** (ou **depth-first**) consiste em visitar a árvore do seguinte modo:

(a) sub-árvore esquerda

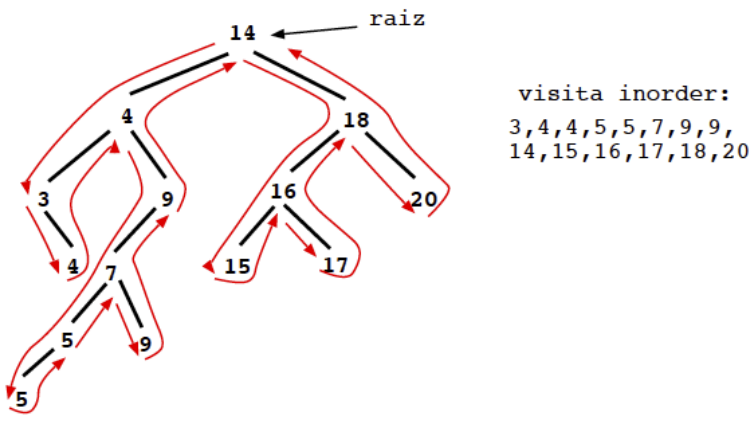
(b) raiz

(c) sub-árvore direita

Implementação da visita inorder

```
public void printInorder() {inorder(root);}
private void inorder(BTNode node) {
    if (node!=null) {
        inorder(node.esq);
        System.out.print(" " + node.val);
        inorder(node.dir);
    }
}
```

A pesquisa inorder equivale a fazer-se o seguinte percurso:



Visitas Preorder e Postorder

Método preorder:

(a) visitar a raiz

(b) pesquisar em preorder a sub-árvore esquerda

(c) pesquisar em preorder a sub-árvore direita

Método postorder:

(a) pesquisar em postorder a sub-árvore esquerda

(b) pesquisar em postorder a sub-árvore direita

(c) visitar a raiz

Exercício: implemente estes métodos de pesquisa.

Árvores Binárias de Pesquisa: procura

Dada uma árvore binária de pesquisa `t` e um valor `x`, pretende-se verificar se `t` contém `x`.
Quando se pensa em "percorrer" uma árvore, deve pensar-se numa solução recursiva em que temos de lidar com três casos:

1. árvore vazia, pelo que a árvore `t` não contém `x`
2. nó corrente contém `x`
3. continuar a procura numa das sub-árvores, de acordo com a relação de ordem entre `x` e o valor no nó corrente.

Árvores Binárias de Pesquisa: procura (2)

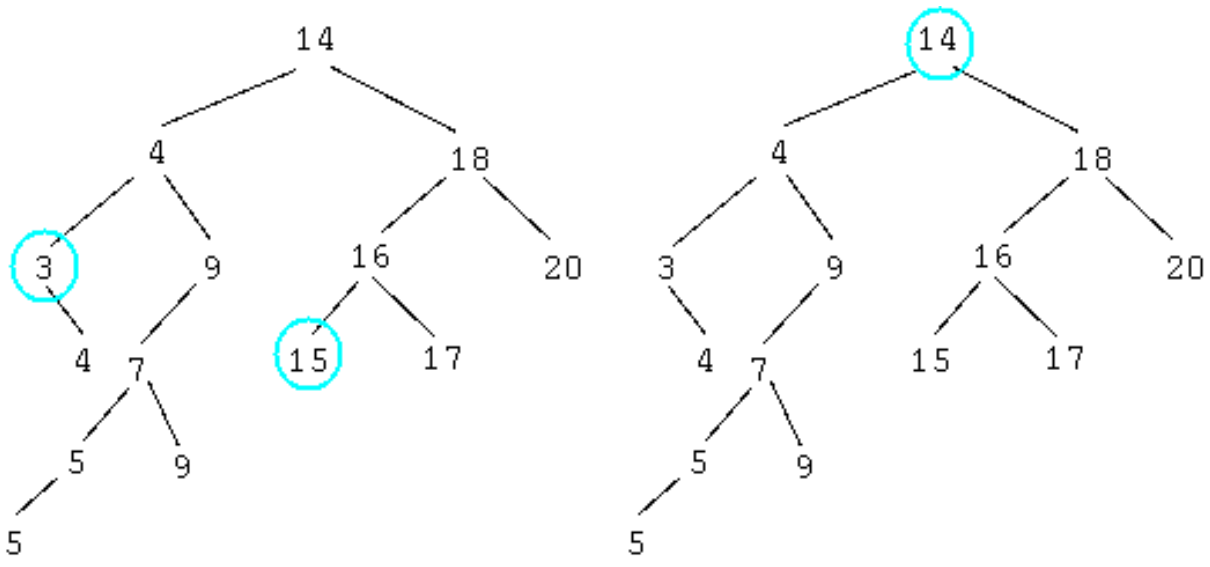
Implementação:

```
public boolean contains(int x) {
    return contains(root, x);
}
private boolean contains(BTNode t, int x) {
    if (t==null)      // caso 1 - arvore vazia
        return false;
    if (x==t.val)     // caso 2 - esta no no corrente
        return true;
    if (x<t.val)      // caso 3 - continua num dos filhos
        return contains(t.esq, x);
    else
        return contains(t.dir, x);
}
```


Árvores Binárias de Pesquisa: remover (1)

Casos especiais para o nó a remover:

- nó folha (e.g. remover o 15)
- nó-interior com apenas um filho (e.g. remover o 3)
- nó interior com dois filhos (e.g. remover o 14)



Árvores Binárias de Pesquisa: remover (2)

A operação de remover um nó que contém um valor `x` de uma árvore `t`, caso tal nó exista, é um pouco mais complexa e requer:

1. localizar o nó com o valor `x`, seja `nx` esse nó;
2. se `nx` é um nó-folha, simplesmente remove-se o nó;
3. se `nx` é um nó-interior, é necessário mais cuidado para não ficarmos com 2 árvores desconexas:
   - se `nx` só tiver um filho, a sub-árvore pendurada nesse nó toma o lugar de `nx`
   - se `nx` tiver dois filhos, então devemos procurar o nó com menor valor entre os descendentes do filho-direito (ou o maior dos descendentes do filho-esquerdo) para tomar o lugar de `nx`.

Árvores Binárias de Pesquisa: remover (3)

```
public void removeBTNode(int x) {
    root= removeBTNode(root, x);
}
private BTNode removeBTNode(BTNode t, int x) {
    if (t!=null) {
        if (x<t.val)
            t.esq= removeBTNode(t.esq, x);
        else if (x>t.val)
            t.dir= removeBTNode(t.dir, x);
        // estamos no nó com o valor a remover
        else if (t.esq==null)
            t= t.dir;
        else if (t.dir==null)
            t= t.esq;
        else          // existem duas sub-árvores
            t.dir= removeMinBTNode(t, t.dir);
    }
    return t;
}
```

Árvores Binárias de Pesquisa: remover (4)

```
private BTNode removeMinBTNode(BTNode d, BTNode t) {
    if (t.esq==null) {
        // t referencia o menor valor
        d.val= t.val;
        return t.dir;
    }
    else
        t.esq= removeMinBTNode(d, t.esq);
    return t;
}
```


Árvores Binárias: inorder não recursivo

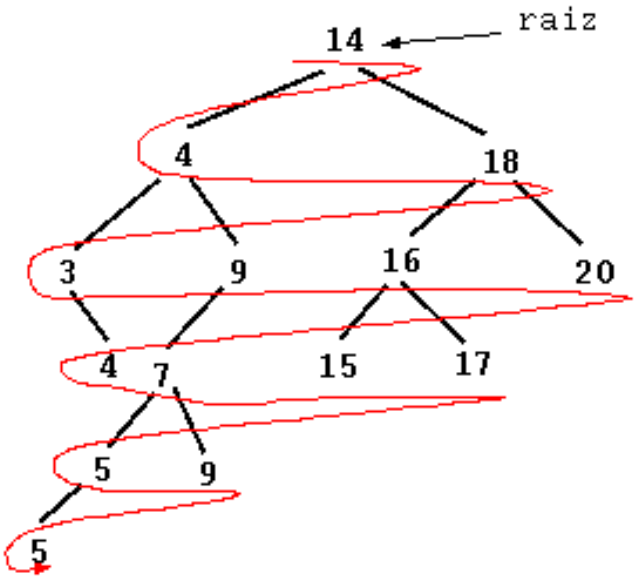
Podemos fazer visitas depth-first (profundidade-primeiro) sem recursão. Precisamos de uma pilha para guardar os nós durante a descida à esquerda de modo a serem recuperados posteriormente para se fazer a descida à direita.

```
private void inorderNR(BTNode r) {
    Stack<BTNode> s= new Stack<BTNode>();
    BTNode t=r; // segunda referencia mesma raiz

    while ((t!=null) || !s.isEmpty()) {
        // seguir os ramos esquerdos primeiro, mas guardar nó-corrente
        while (t!=null) {
            s.push(t);
            t= t.esq;  // visita sub-árvore esquerda
        }
        if (!s.isempty()) {
            t= s.pop();
            System.out.print(" " + t.val);
            t= t.dir;  // visita sub-árvore direita
        }
    }
}
```

Pesquisa em largura-primeiro (breadth-first)

Um outro método de percorrer os nós de uma árvore é fazê-lo por níveis, i.e. em largura-primeiro (ou breadth-first).



visita breadth-first:
14,4,18,3,9,16,20,
4,7,15,17,5,9,5

Árvores Binárias de Pesquisa: breadthFirst

A implementação deste algoritmo faz uso de uma fila onde se colocam os filhos do nó corrente. A fila garante que a ordem de visita no nível seguinte é preservada.

```
private void breadthFirst(BTNode r) {
    Queue<BTNode> fila= new Queue<BTNode>();
    BTNode t;

    fila.add(r);
    while (!fila.isEmpty()) {
        t= fila.remove();
        System.out.print(" " + t.info);
        if (t.esq!=null) fila.add(t.esq);
        if (t.dir!=null) fila.add(t.dir);
    }
}
```

Exercício com árvores: espelho (1)

Objectivo: escrever um método em que dada uma árvore produza o espelho dessa árvore.

- Por espelho de uma árvore entenda-se uma outra árvore em que os *papeis* das referências para o filho-esquerdo e filho-direito estão trocadas em todos os nós.
- A figura ilustra 2 árvores, onde a 2a. é o espelho da 1a.

Exercício com árvores: espelho (2)

```
// mirror de uma arvore com root r
BTNode mirror(BTNode r) {
    if (r== null)
        return null;
    BTNode tmp;

    // faz o mirror e troca refs
    tmp= mirror(r.esq);
    r.esq= mirror(r.dir);
    r.dir= tmp;
    return r;
}
```

Ver solução completa na página dos apontamentos.