



PÓS-GRADUAÇÃO  
COMPUTAÇÃO APLICADA

# CAP241 – Computação Aplicada I

## Estruturas de Dados Elementares

### Parte III – Árvores

**Gilberto Ribeiro de Queiroz**

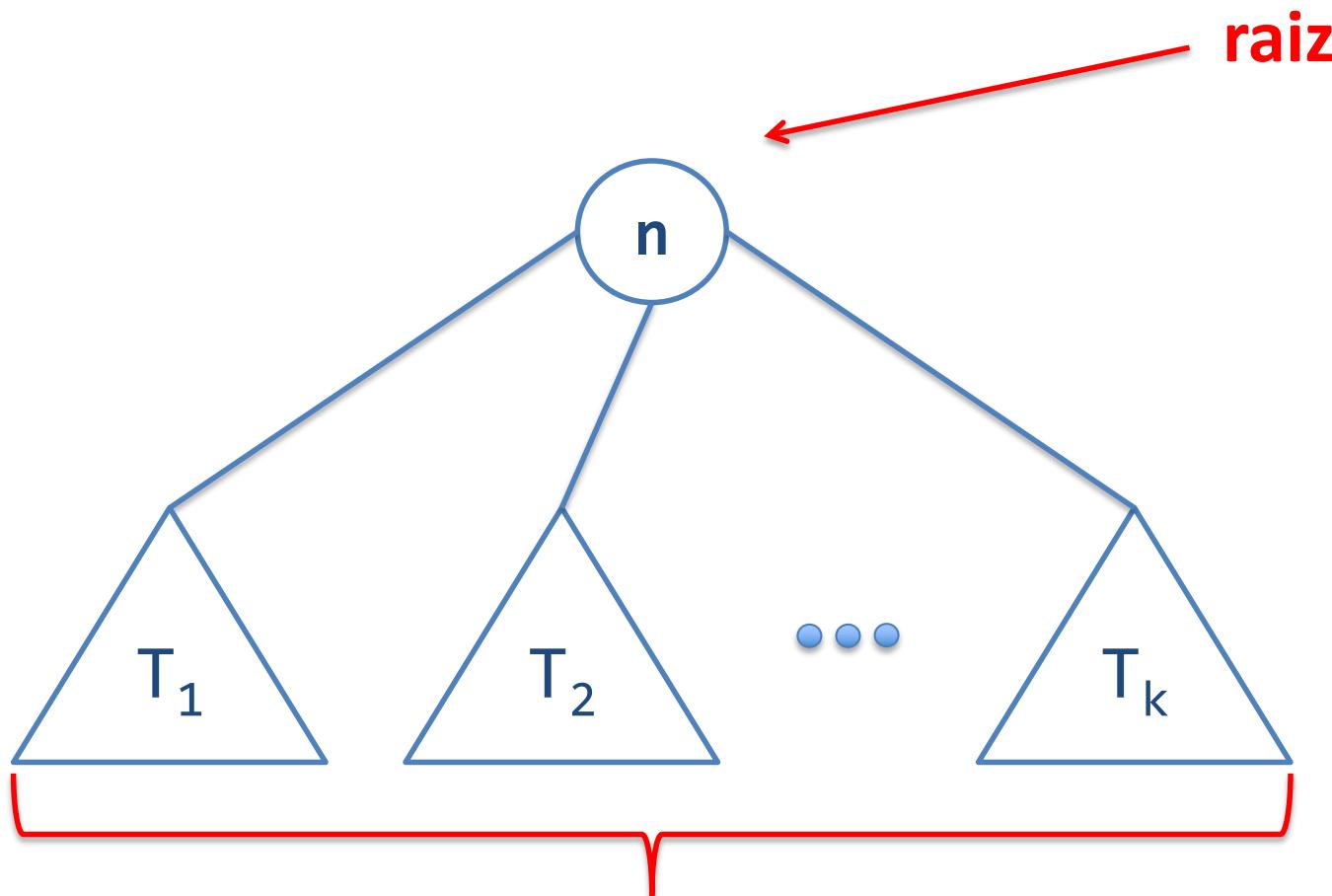
Instituto Nacional de Pesquisas Espaciais (INPE) – Brasil  
Ciência da Computação, PhD  
P&D em Geocomputação

# Visão Geral da Aula

- Conceito de Árvores e suas Representações
- Árvores Binárias
- Travessia em Árvores Binárias
- Árvores Binárias de Pesquisa
- Contêineres Associativos da Biblioteca STL
- Considerações Finais

# Árvores

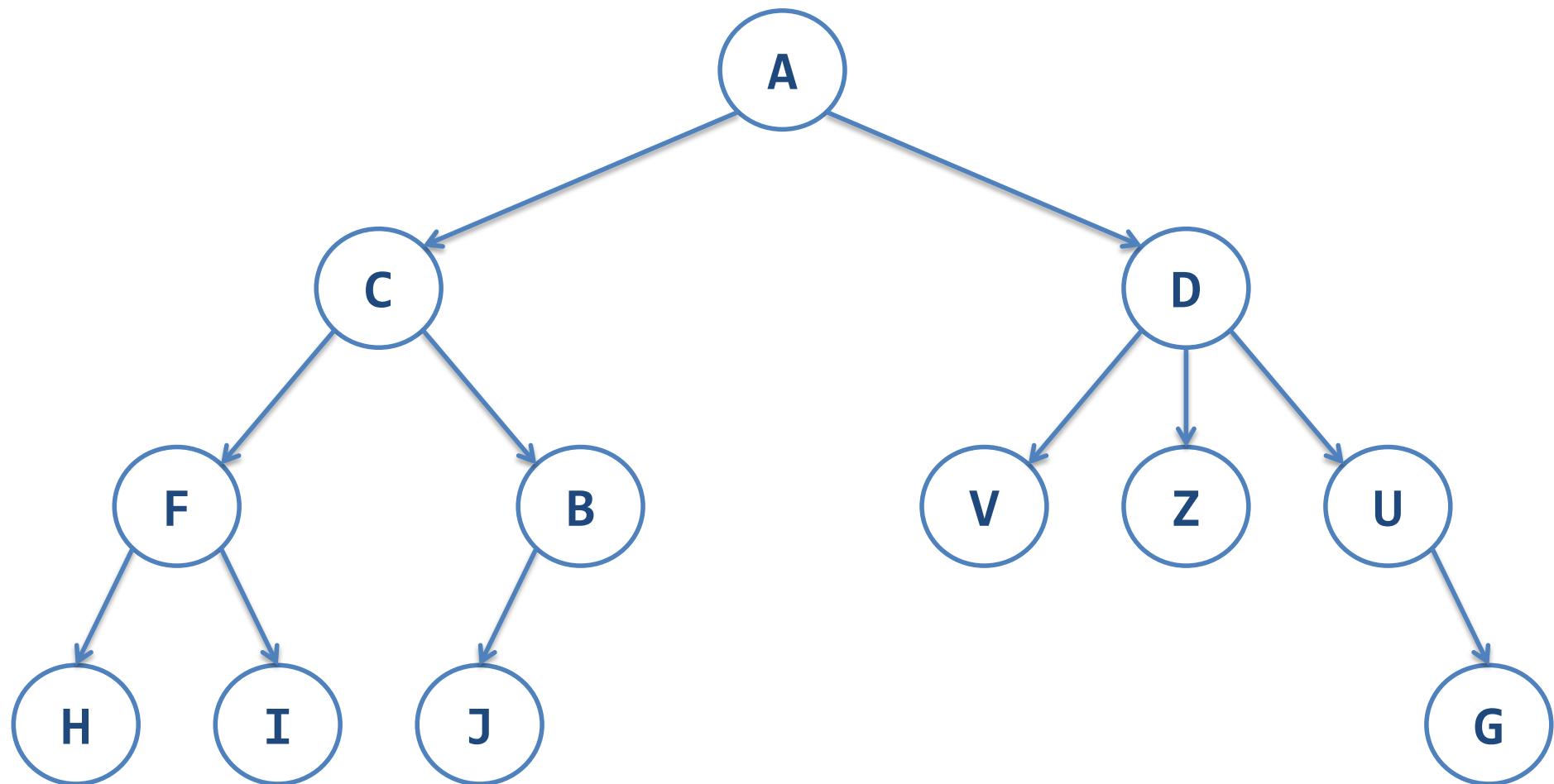
# Definição: Árvores com Raiz



**k conjuntos disjuntos,  $k \geq 0$ , que  
também são árvores**

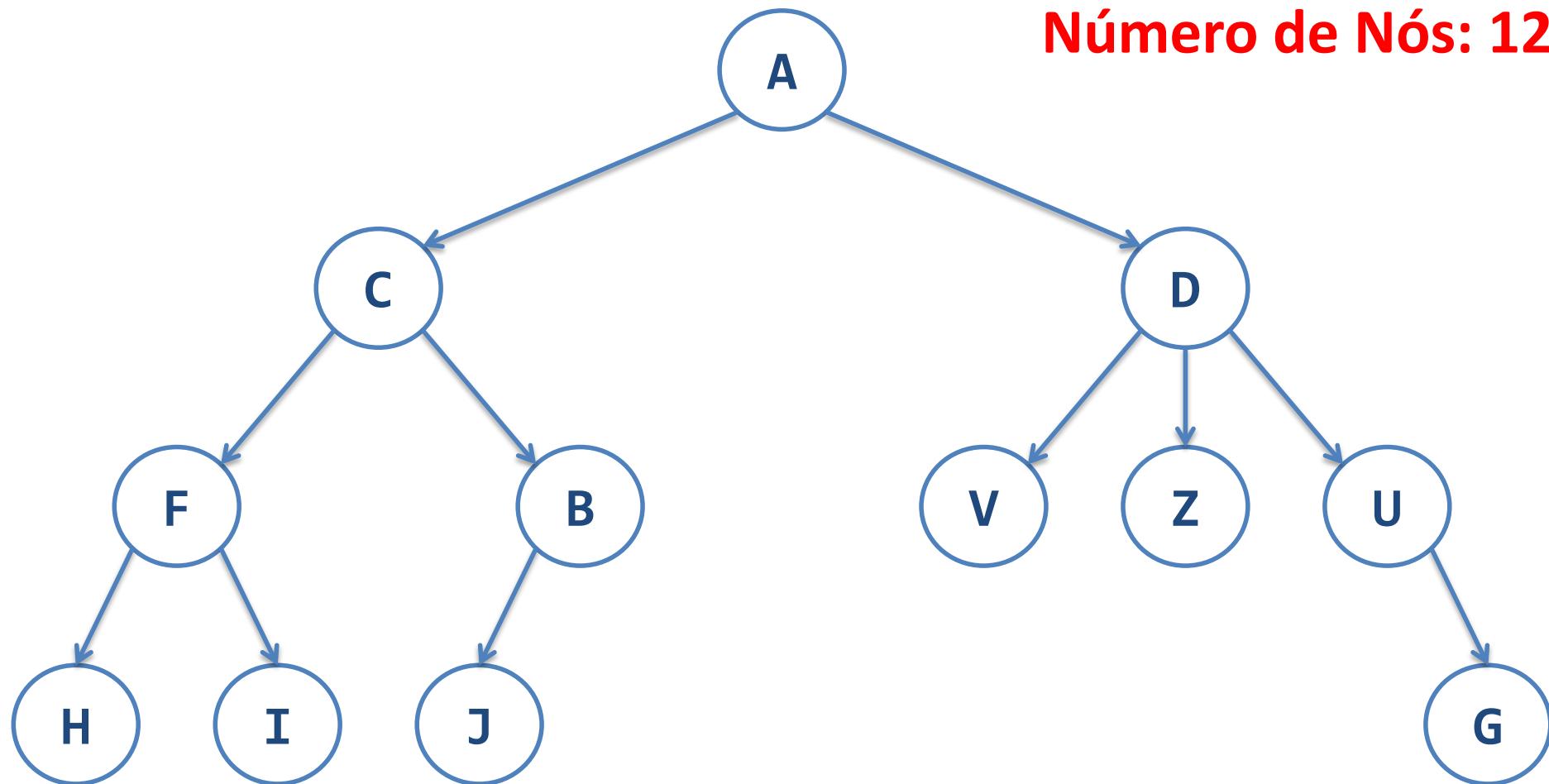
# Árvores: Terminologia

# Árvores: Terminologia

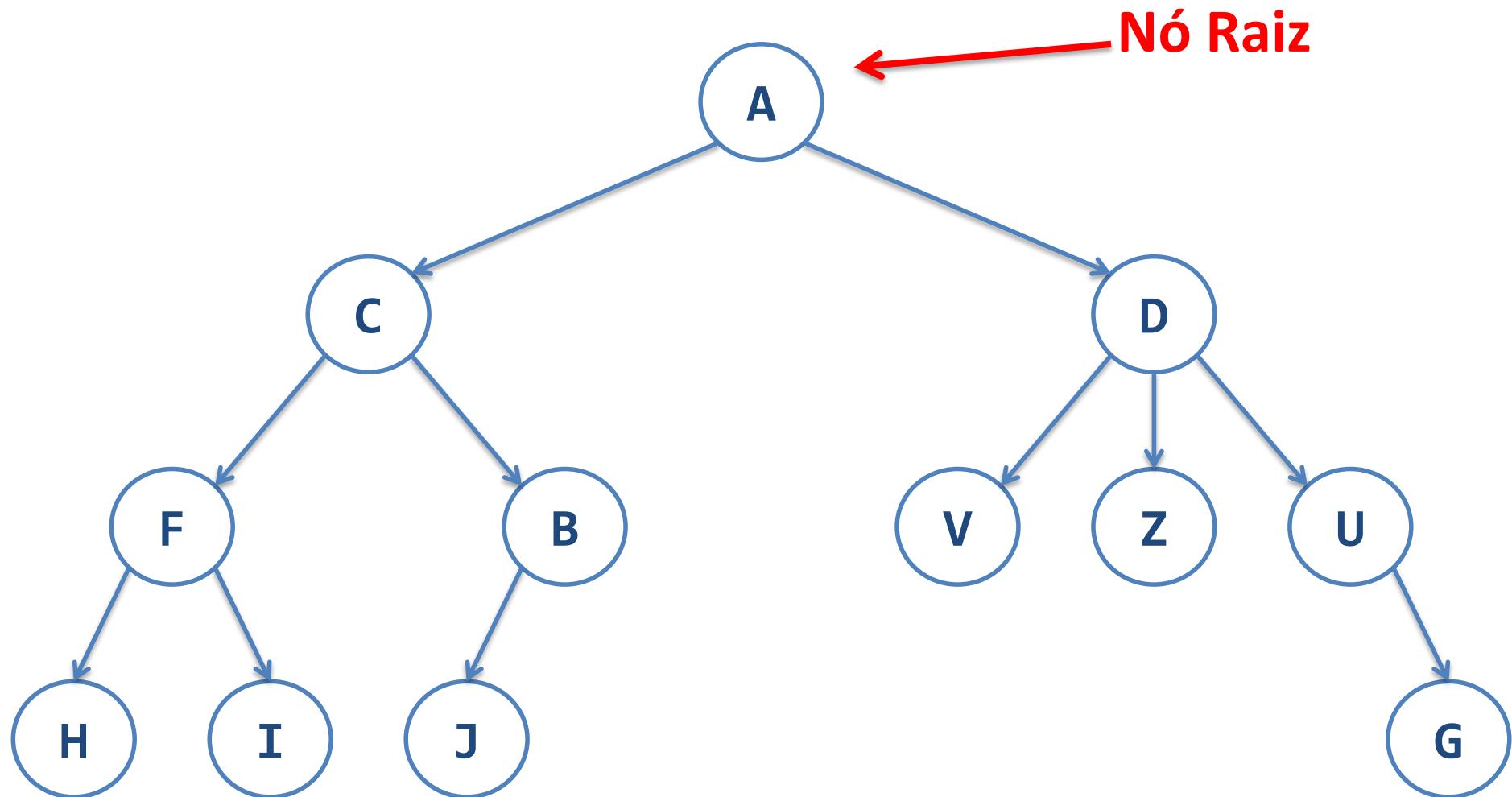


# Árvores: Terminologia

Número de Nós: 12

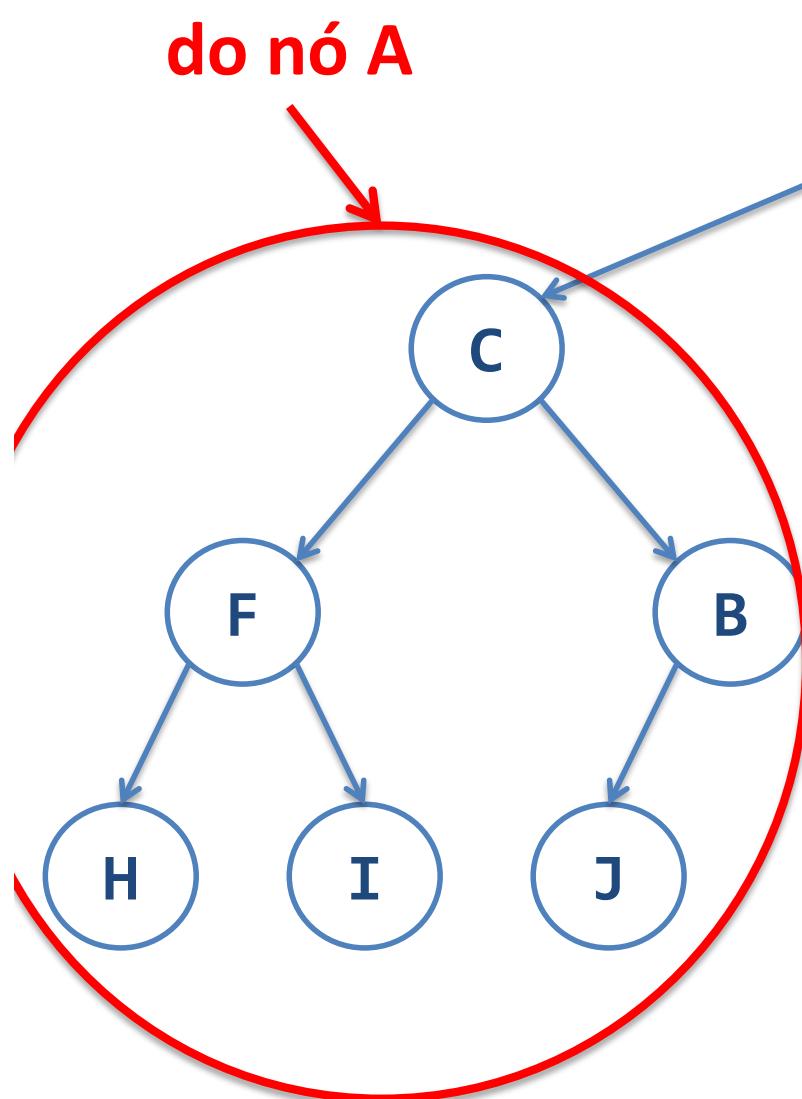


# Árvores: Terminologia

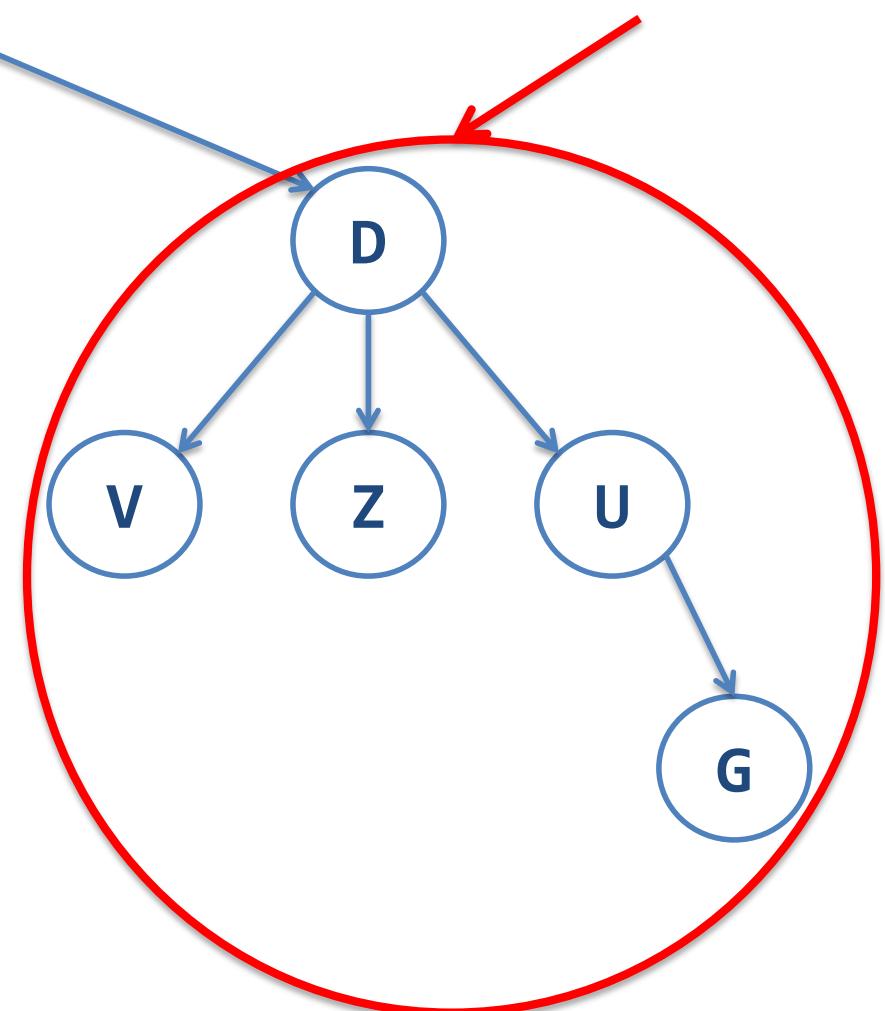


# Árvores: Terminologia

Sub-árvore  
do nó A

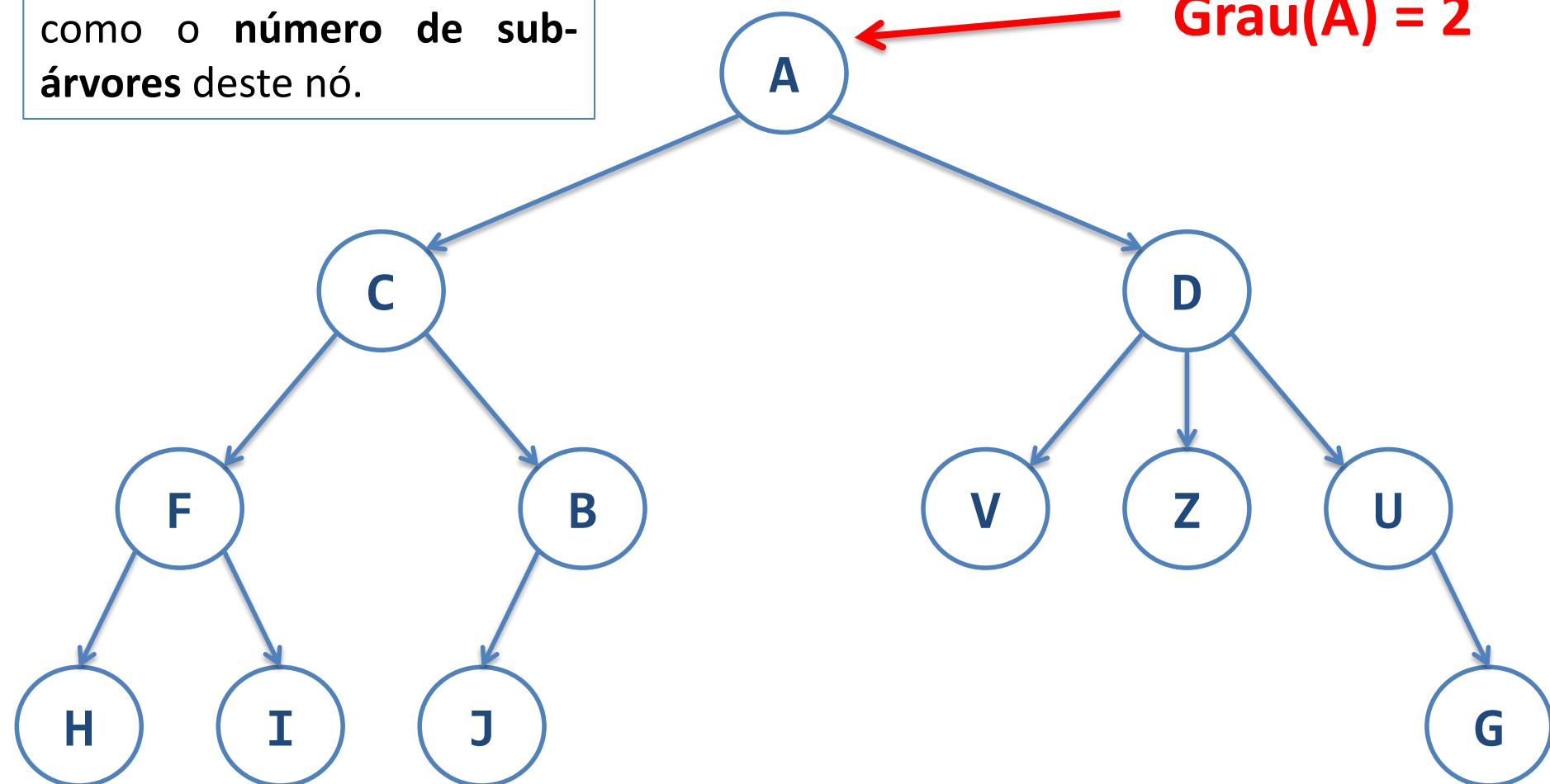


Sub-árvore  
do nó A



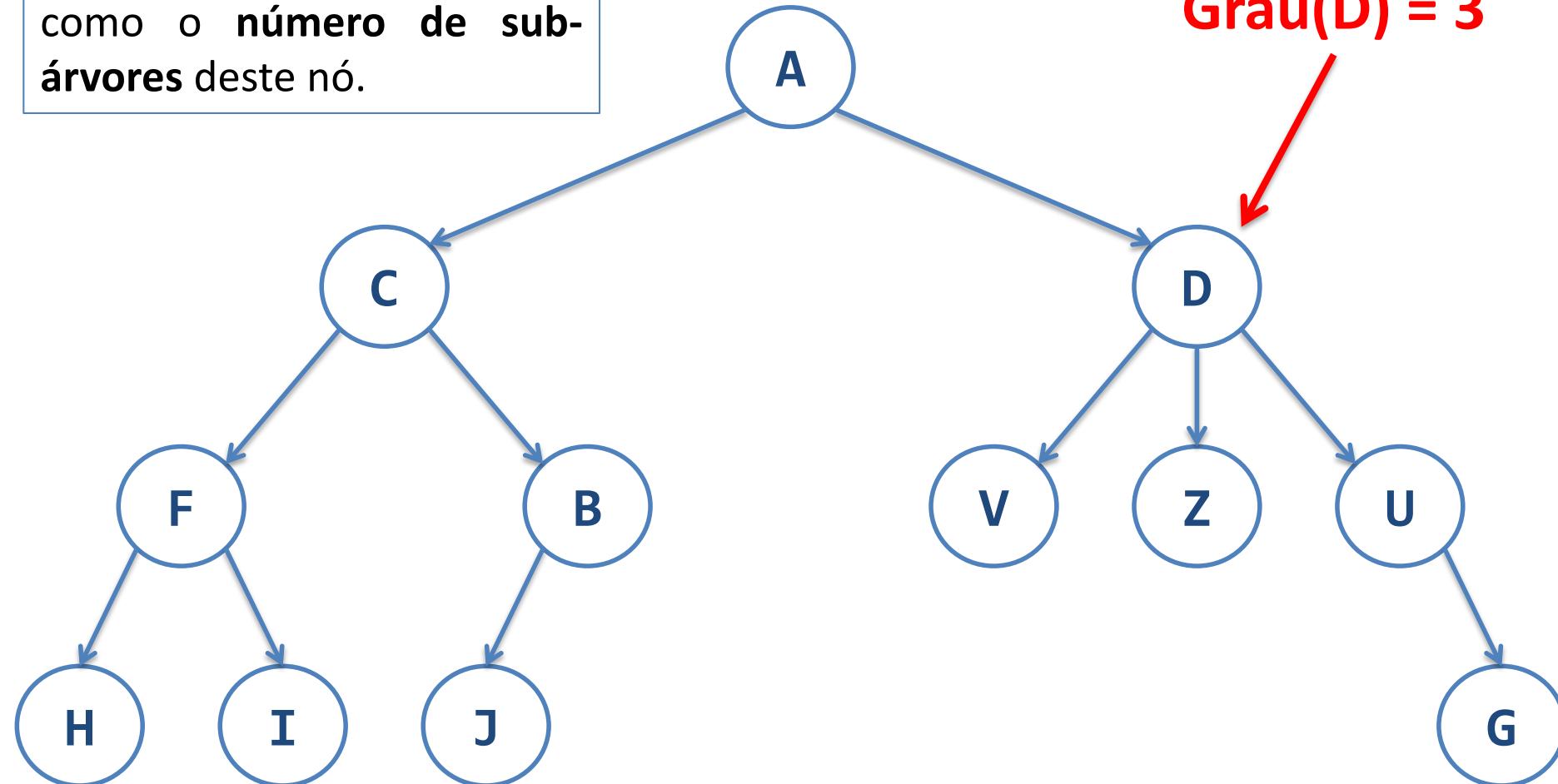
# Árvores: Terminologia

O grau de um nó é definido como o número de sub-árvore desse nó.



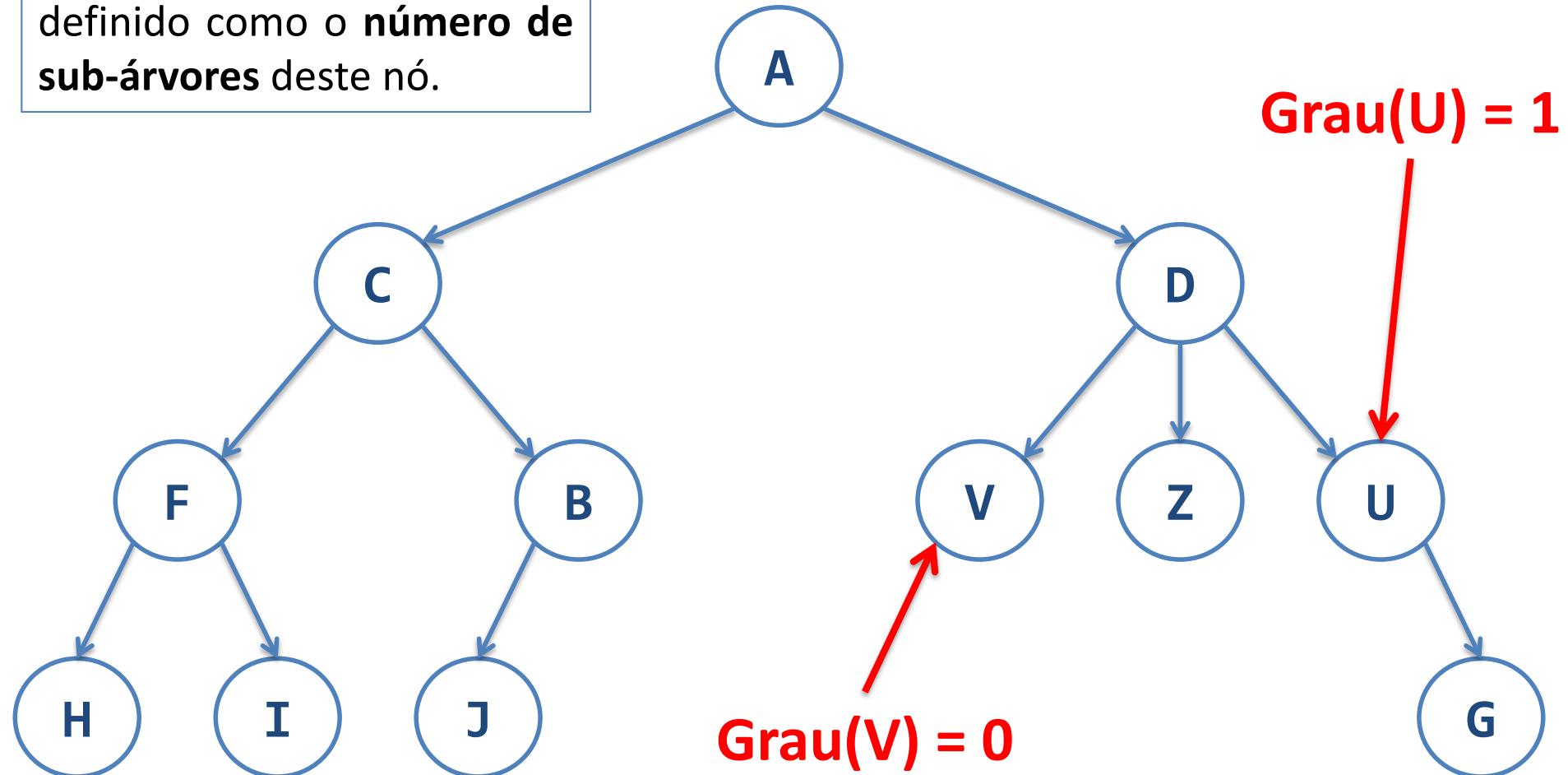
# Árvores: Terminologia

O grau de um nó é definido como o número de sub-árvore desse nó.



# Árvores: Terminologia

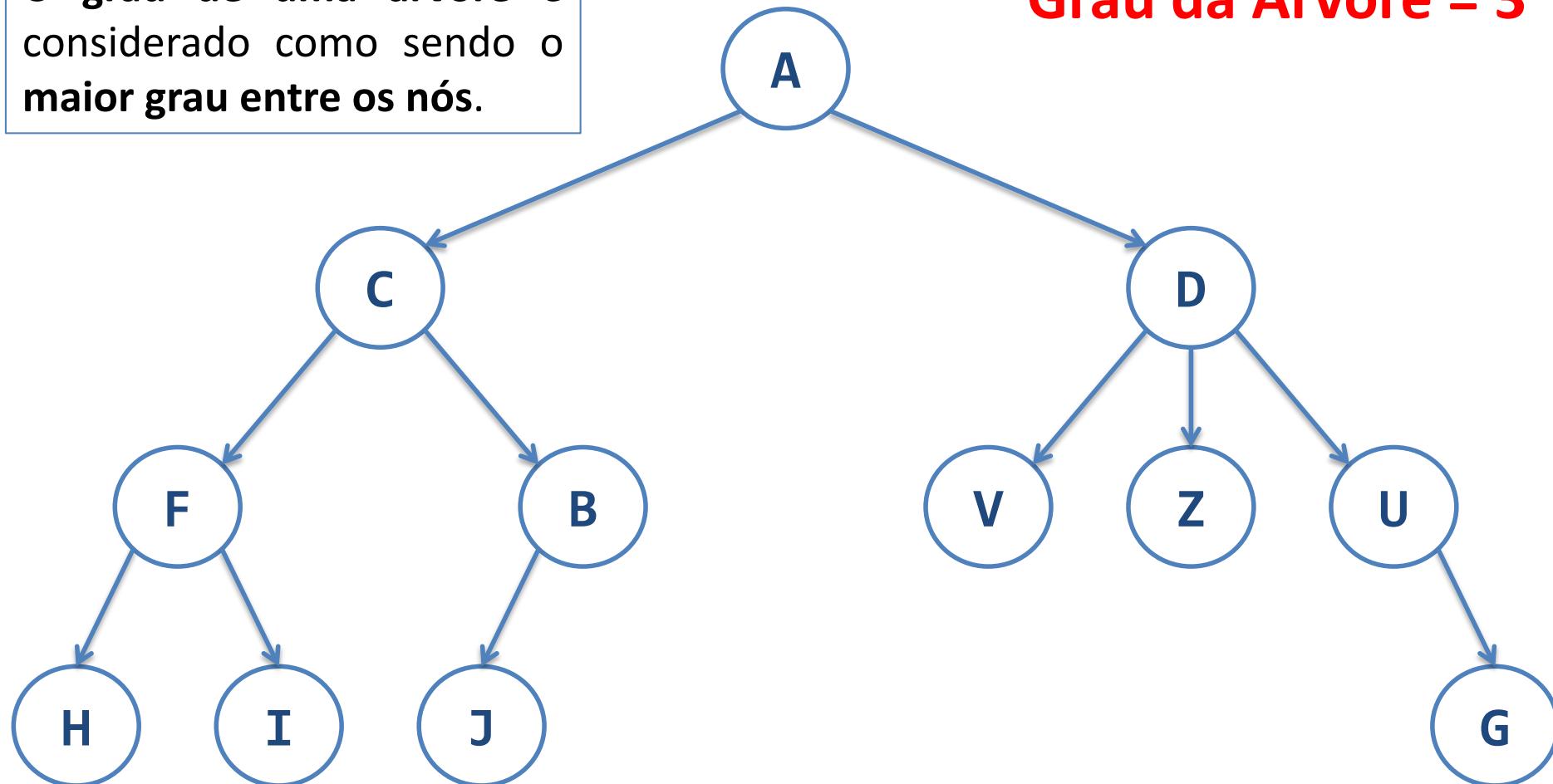
O grau de um nó (*degree*) é definido como o número de sub-árvores deste nó.



# Árvores: Terminologia

O grau de uma árvore é considerado como sendo o maior grau entre os nós.

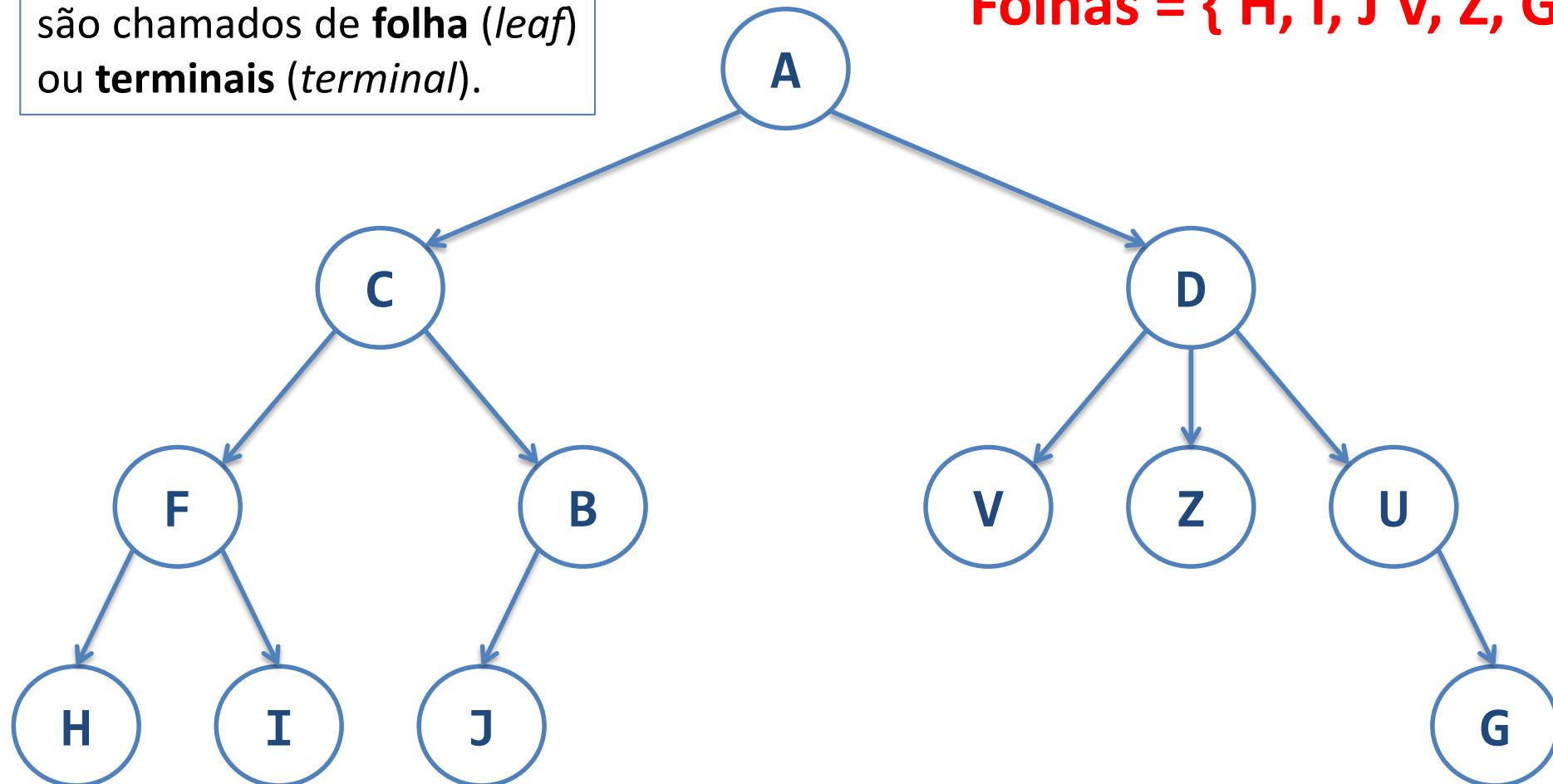
**Grau da Árvore = 3**



# Árvores: Terminologia

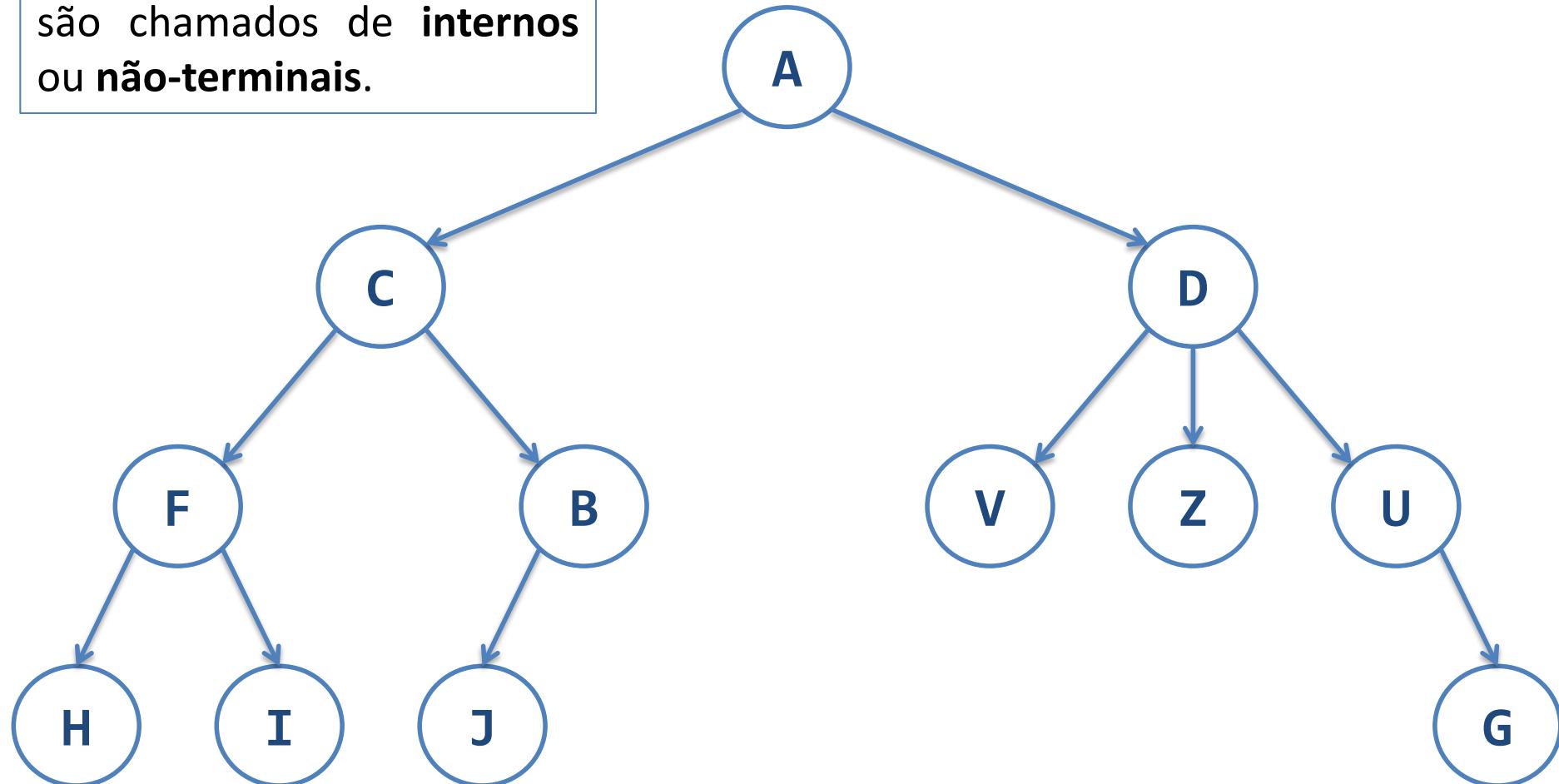
Os nós que possuem grau 0  
são chamados de folha (*leaf*)  
ou terminais (*terminal*).

**Folhas = { H, I, J, V, Z, G }**



# Árvores: Terminologia

Os nós que não são folha, são chamados de **internos** ou **não-terminais**.

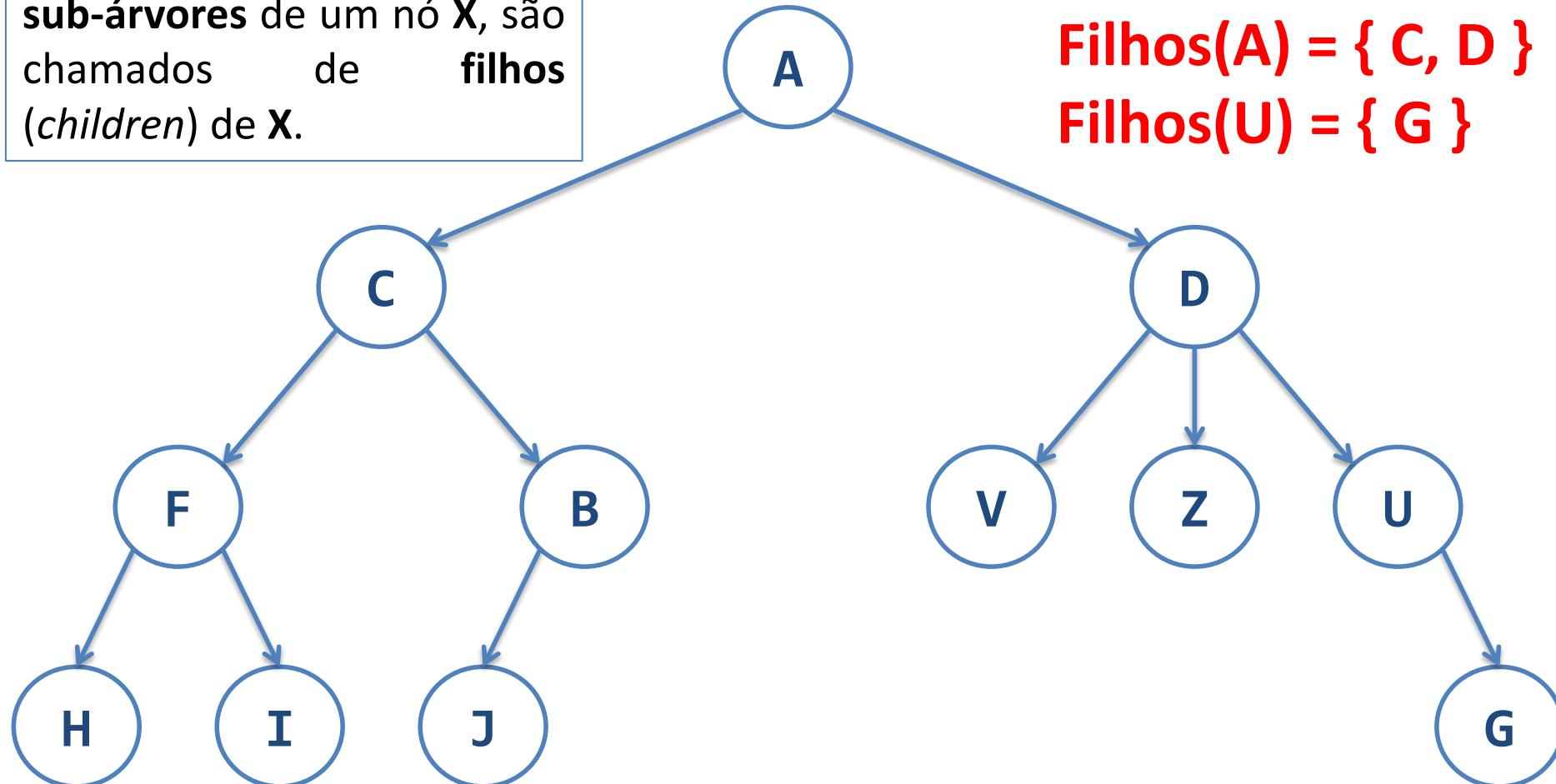


**Não-Terminais = { A, C, D, F, B, U }**

# Árvores: Terminologia

Os nós que são raízes das sub-árvores de um nó X, são chamados de filhos (*children*) de X.

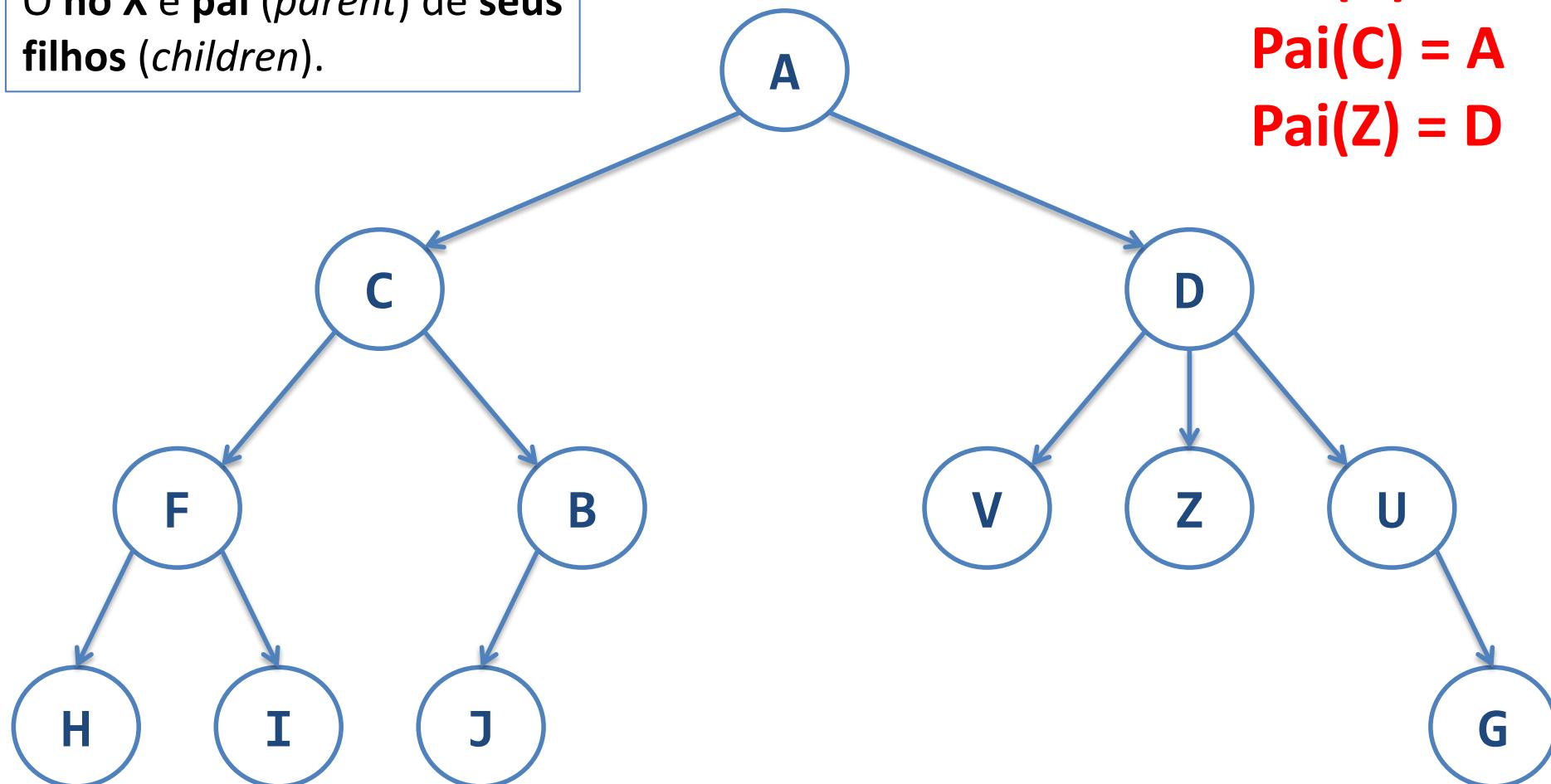
**Filhos(D) = { V, Z, U }**  
**Filhos(A) = { C, D }**  
**Filhos(U) = { G }**



# Árvores: Terminologia

O nó X é pai (*parent*) de seus filhos (*children*).

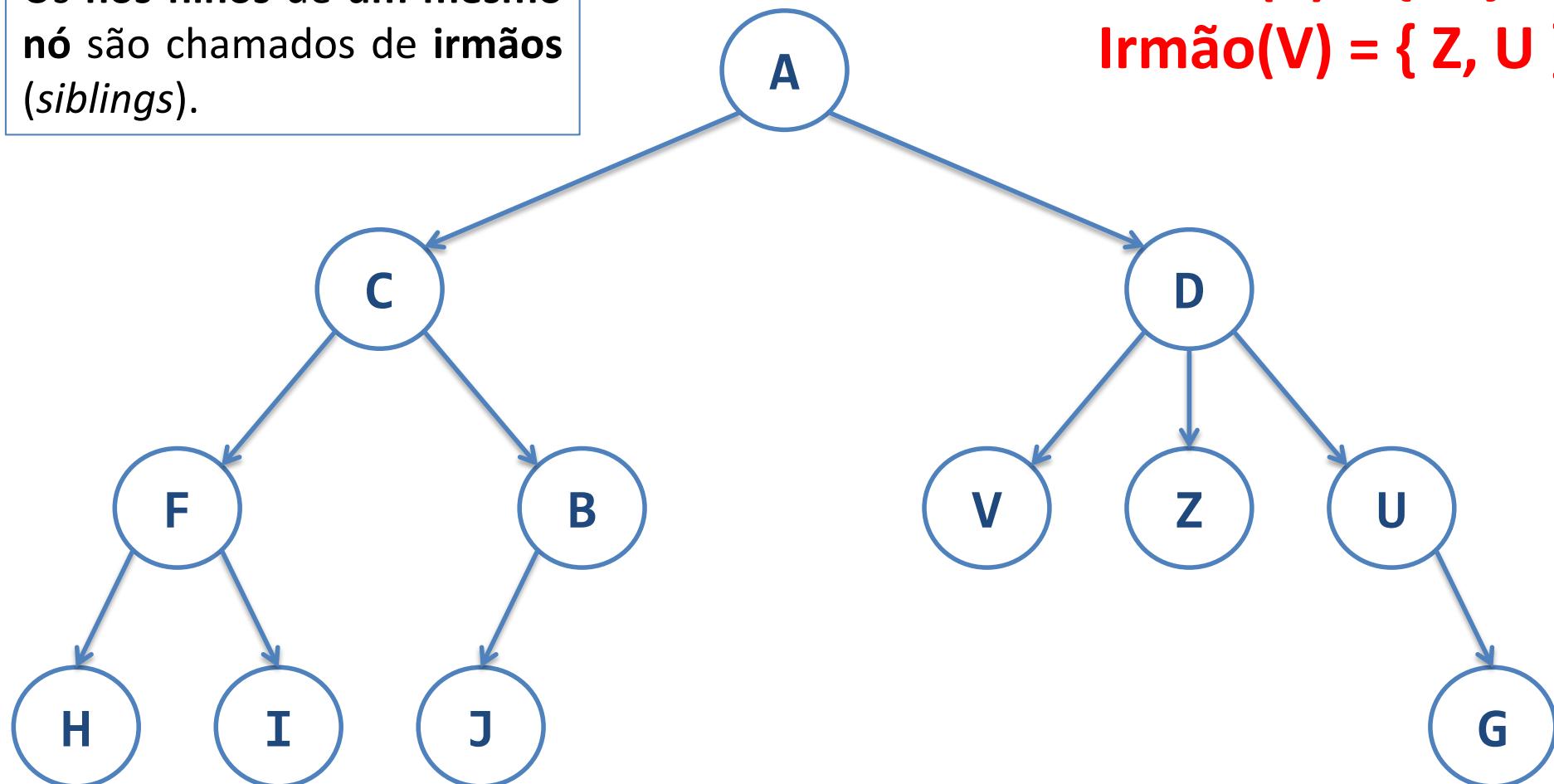
**Pai(D) = A**  
**Pai(C) = A**  
**Pai(Z) = D**



# Árvores: Terminologia

Os nós filhos de um mesmo nó são chamados de irmãos (*siblings*).

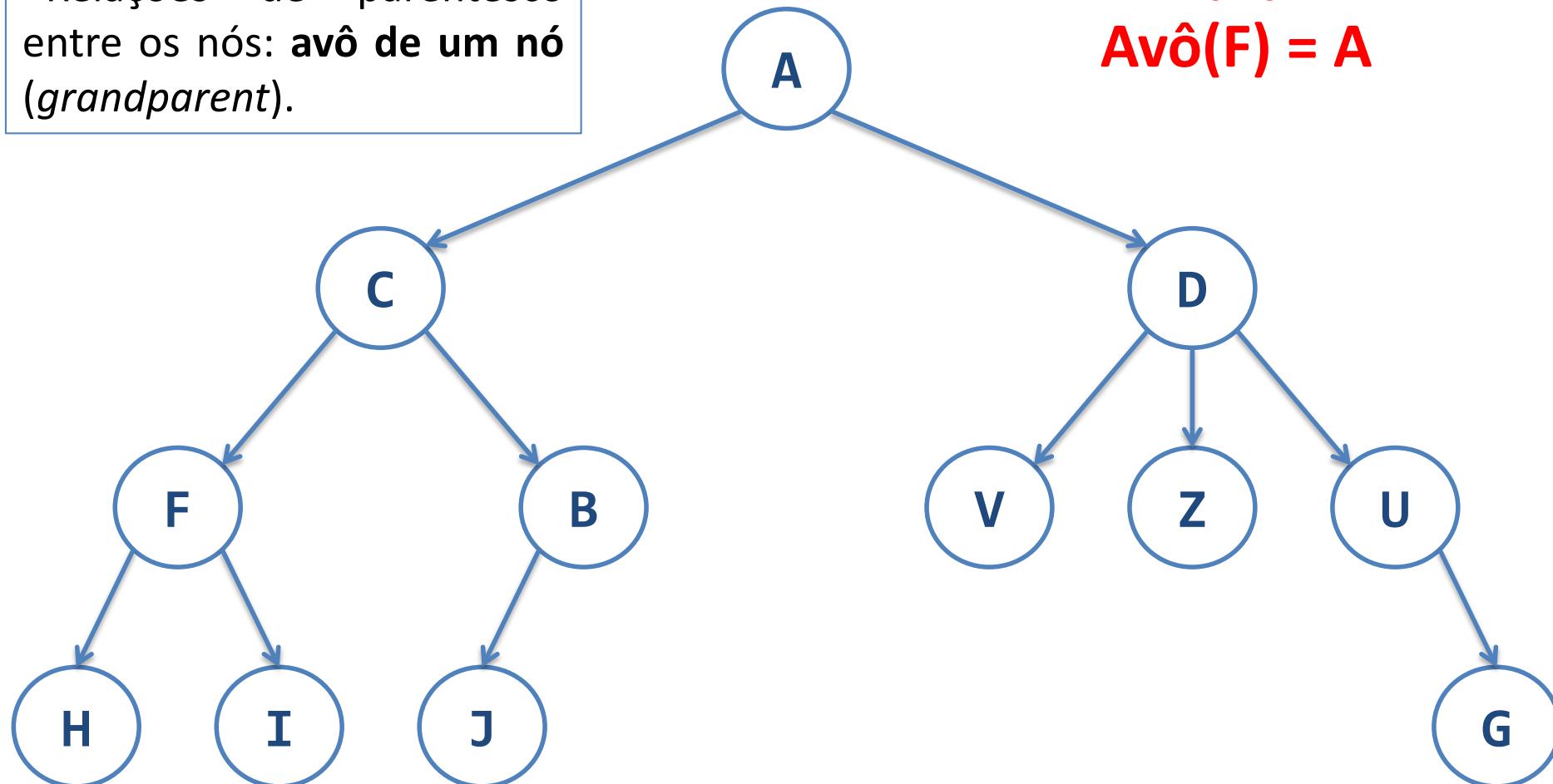
Irmão(C) = { D }  
Irmão(V) = { Z, U }



# Árvores: Terminologia

+Relações de parentesco entre os nós: **avô de um nó** (*grandparent*).

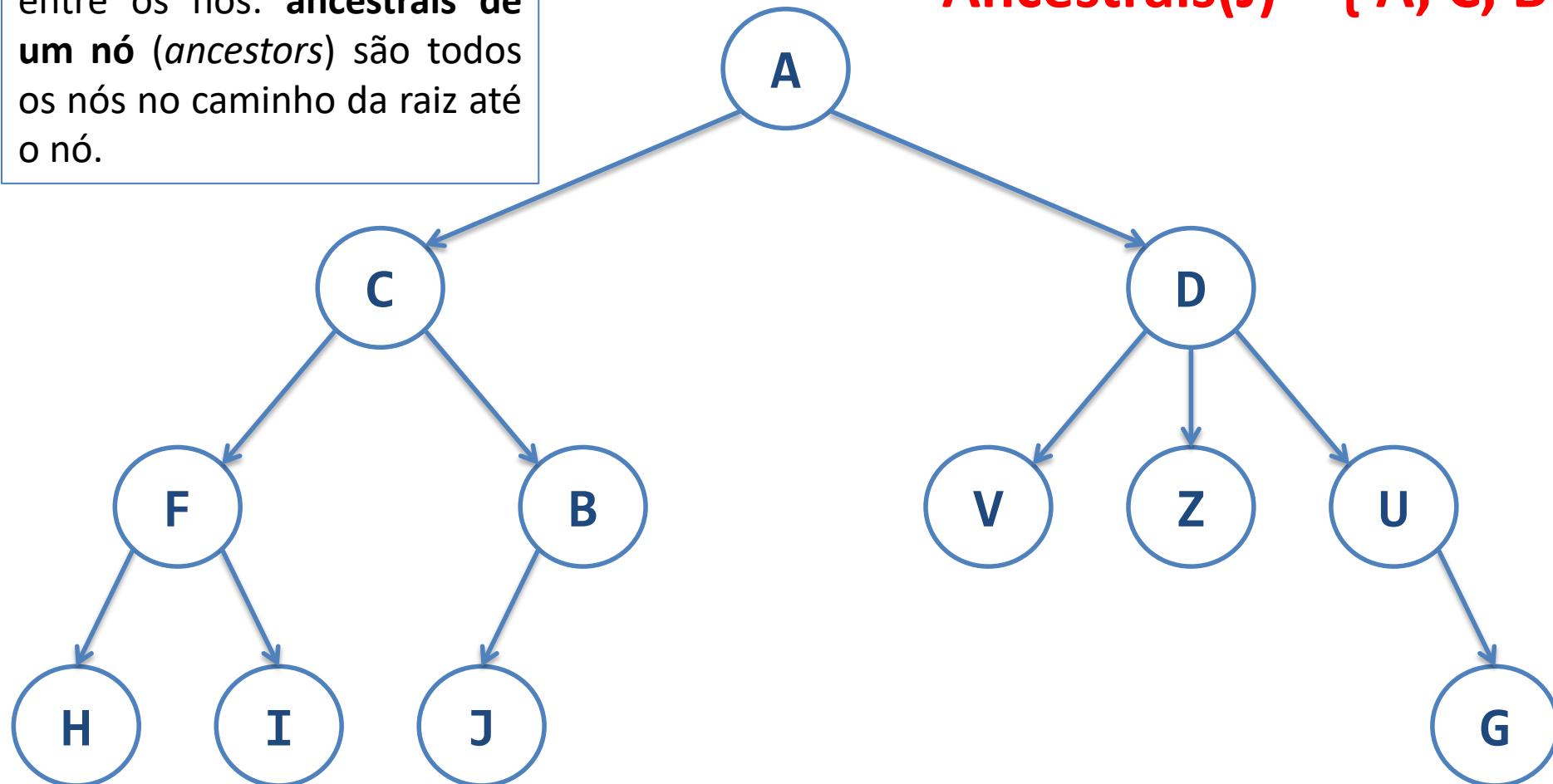
$$\text{Avô}(G) = D$$
$$\text{Avô}(F) = A$$



# Árvores: Terminologia

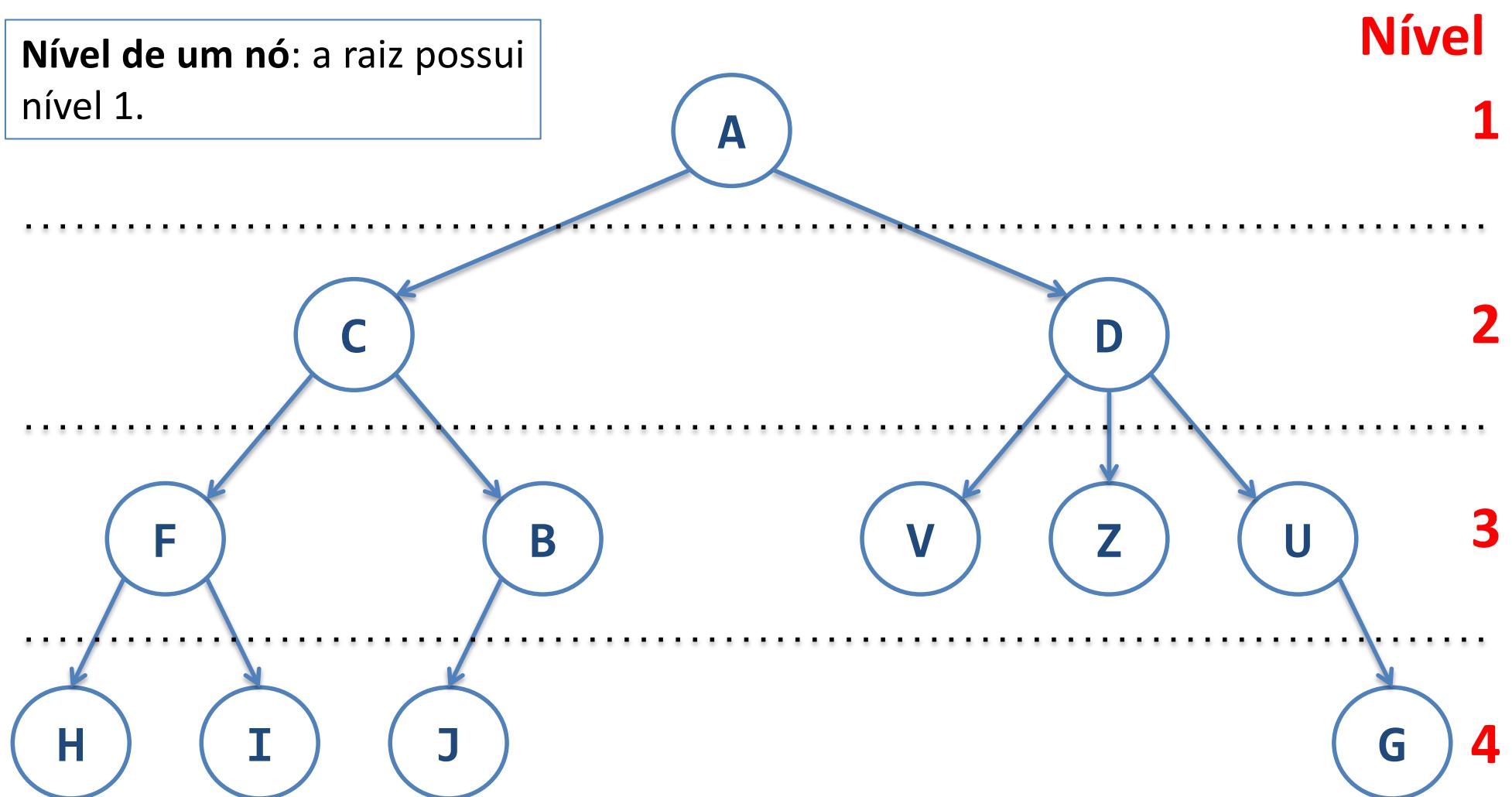
+Relações de parentesco entre os nós: **ancestrais de um nó (ancestors)** são todos os nós no caminho da raiz até o nó.

**Ancestrais(J) = { A, C, B }**



# Árvores: Terminologia

**Nível de um nó:** a raiz possui nível 1.

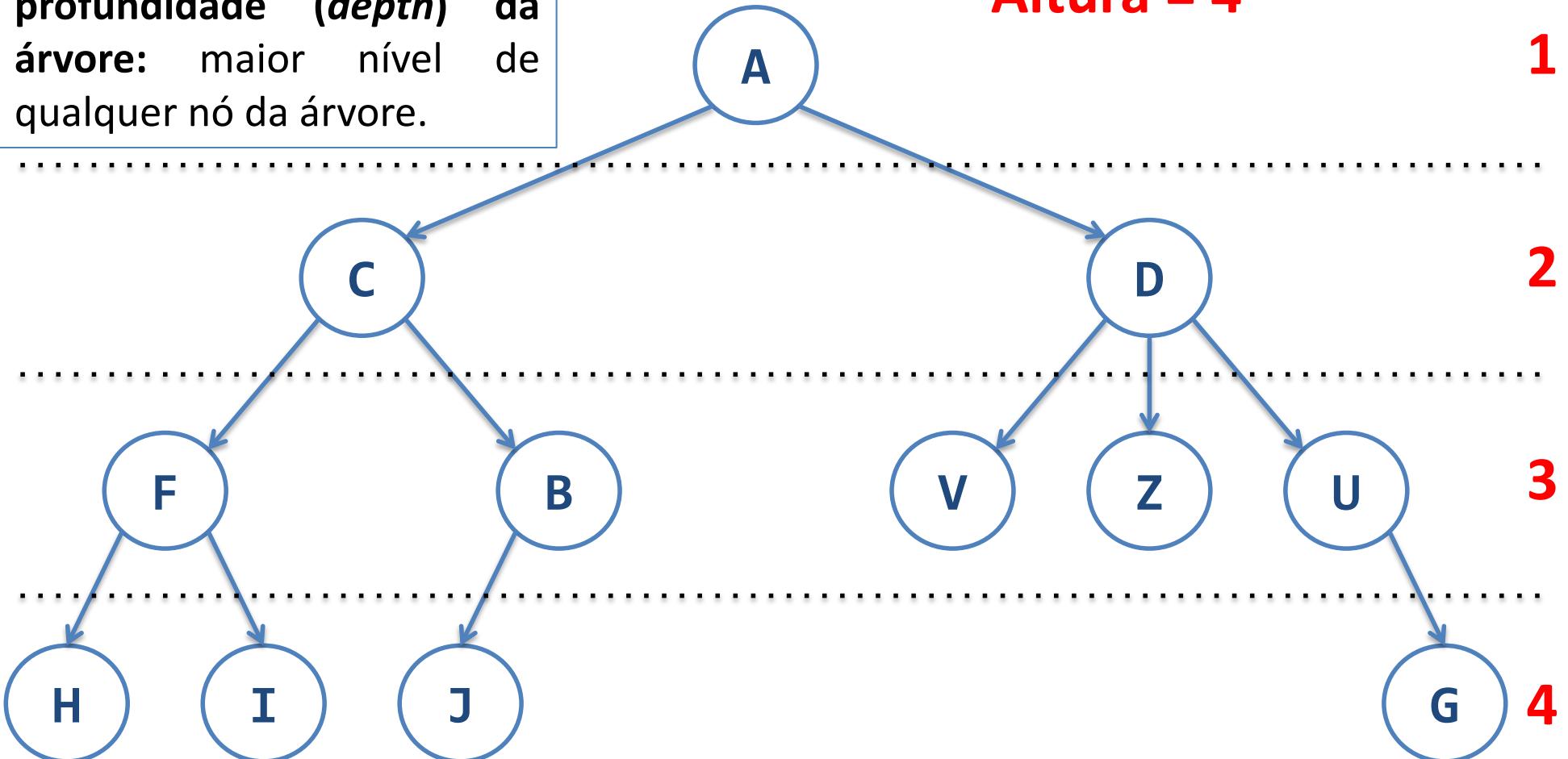


# Árvores: Terminologia

Altura (*height*) ou profundidade (*depth*) da árvore: maior nível de qualquer nó da árvore.

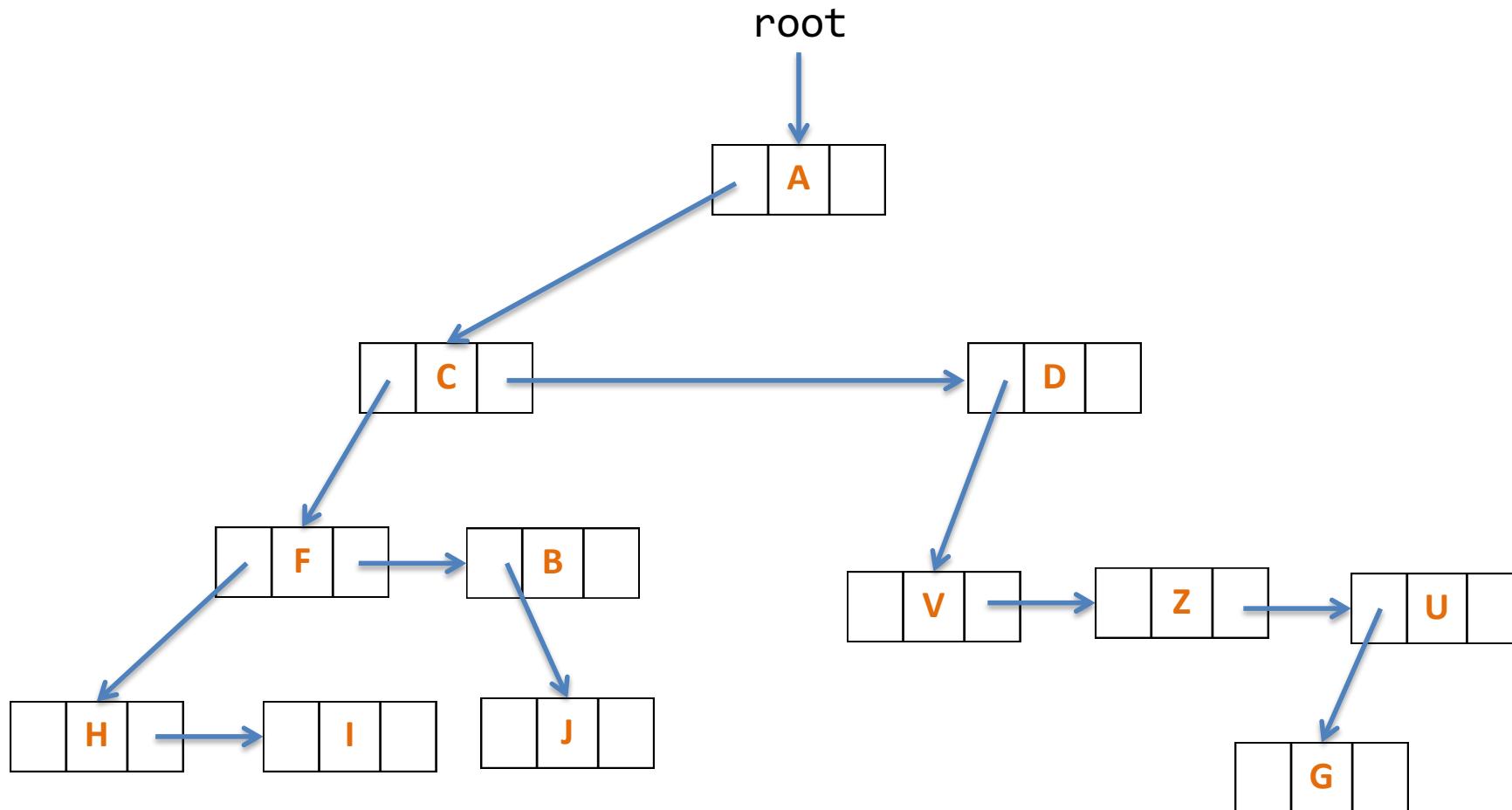
Altura = 4

Nível  
1



# Representação por meio de Listas

# Representação por meio de Listas



# Árvores Multivias

# Árvores Multivias

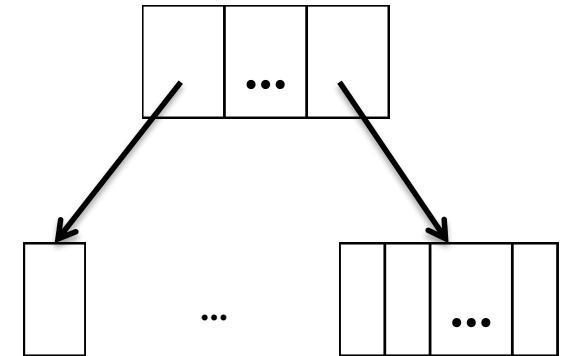
- Árvores com grau máximo maior do que 2 são conhecidas por **multivias**.
- O número máximo de nós **n** em uma árvore **multivias** de grau **d** e altura **h** é dado por:

$$n = 1 + d + d^2 + \dots + d^{h-1} = \sum_{i=0}^{h-1} d^i = \frac{1 - d^{h-1}}{1 - d}, h > 0$$

Usando a STL, como poderíamos  
representar uma árvore multivias de  
maneira simples?

# Uma Árvore Multivias com a STL

```
template<class T> class tree
{
public:
    tree(tree *parent = nullptr);
    ...
private:
    tree *parent_;
    std::vector<tree*> children_;
    T data_;
};
```



# Uma Árvore Multivias com a STL

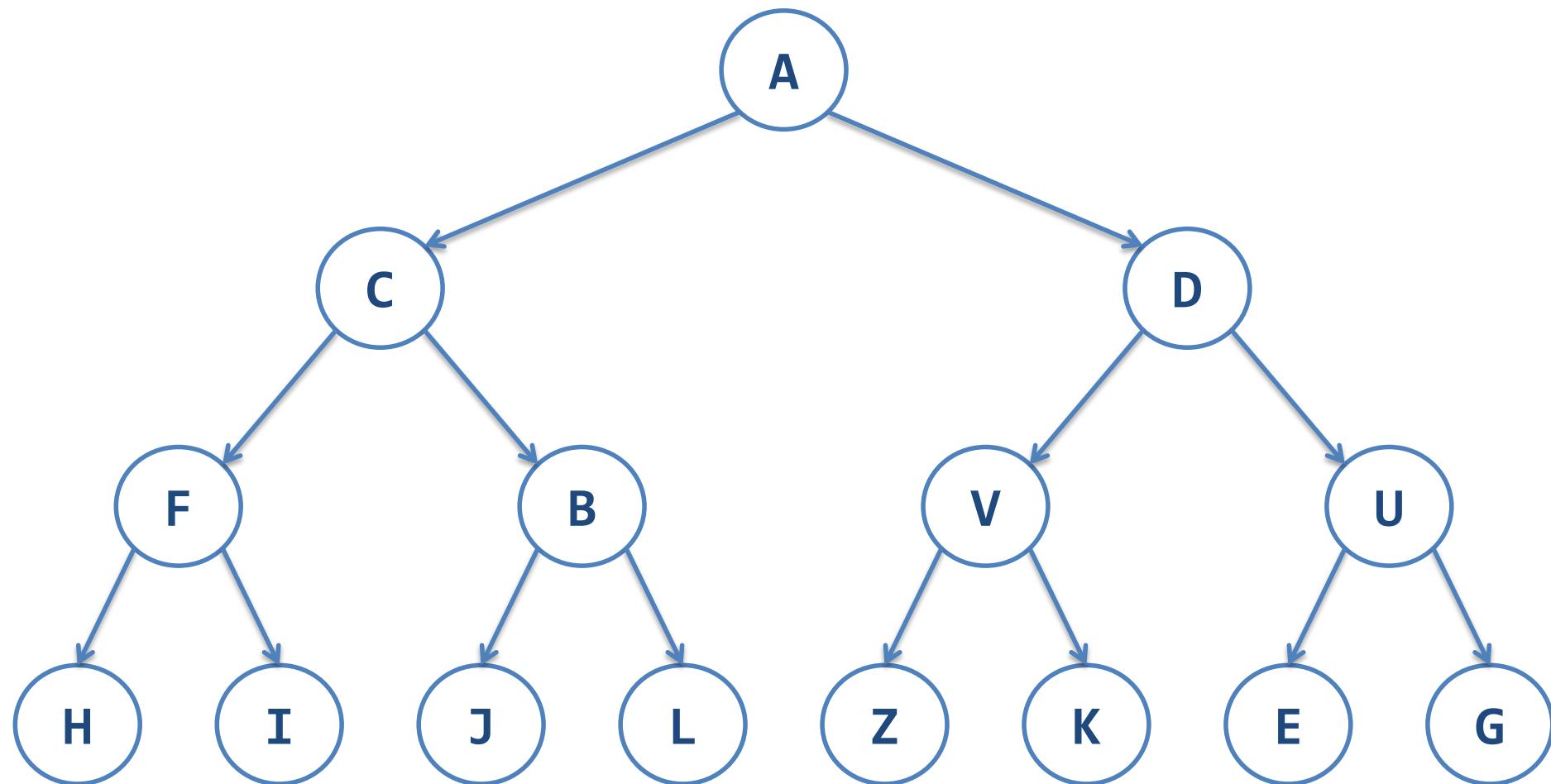
- Operações interessantes que poderíamos acrescentar à classe **tree**:

```
tree* get_parent() const  
bool has_children() const  
size_t get_children_count() const  
size_t get_descendants_count() const  
tree* get_child(size_t i) const  
void push_back(tree *child)  
void insert(std::size_t i, tree *child)  
unique_ptr<tree> erase(size_t i)  
std::size_t get_index() const  
bool is_sibling(const tree* item) const  
void disconnect()  
...
```

# Árvores Binárias

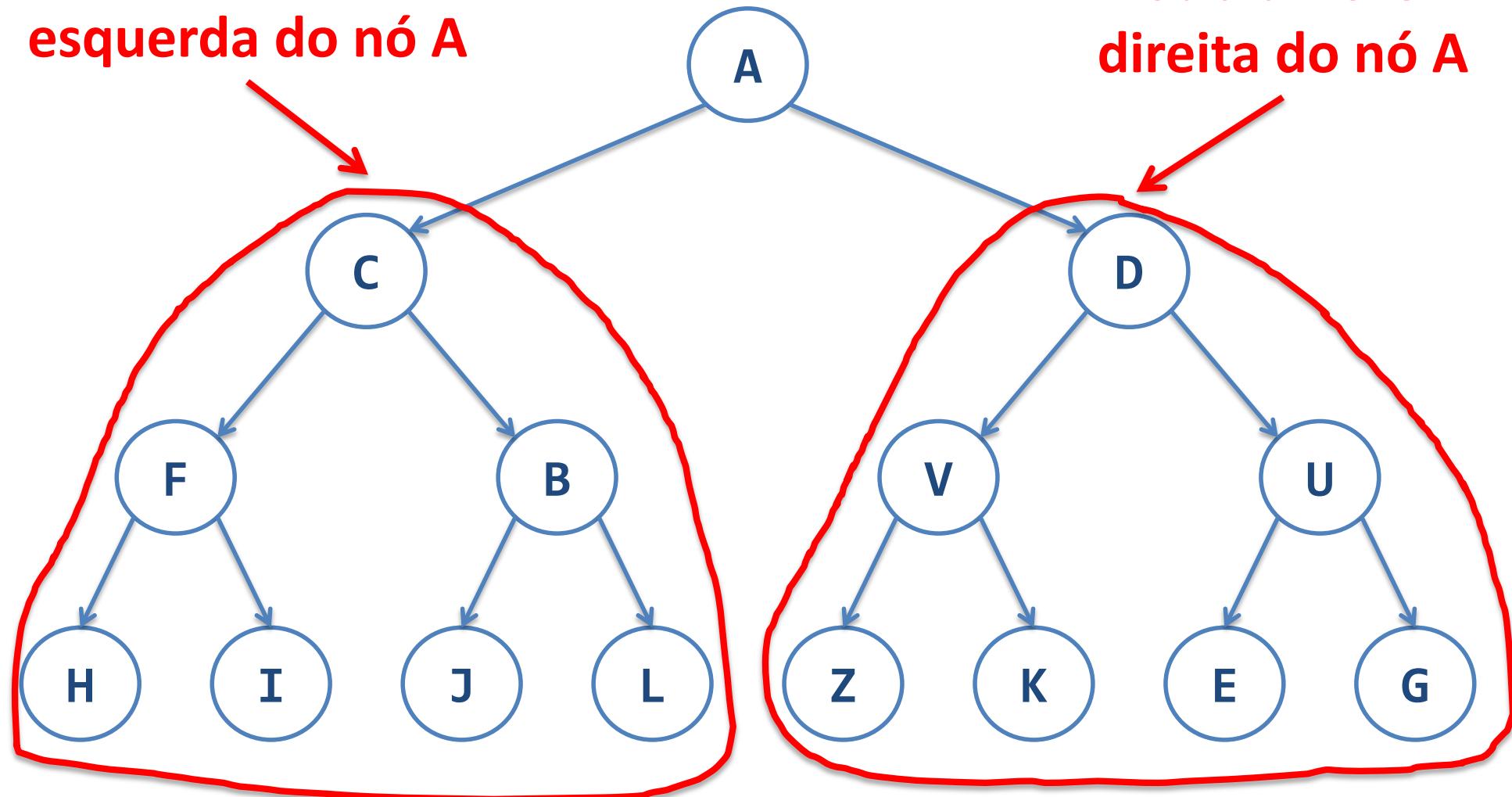
## *(Binary Trees)*

# Árvores Binárias: Grau Máximo 2



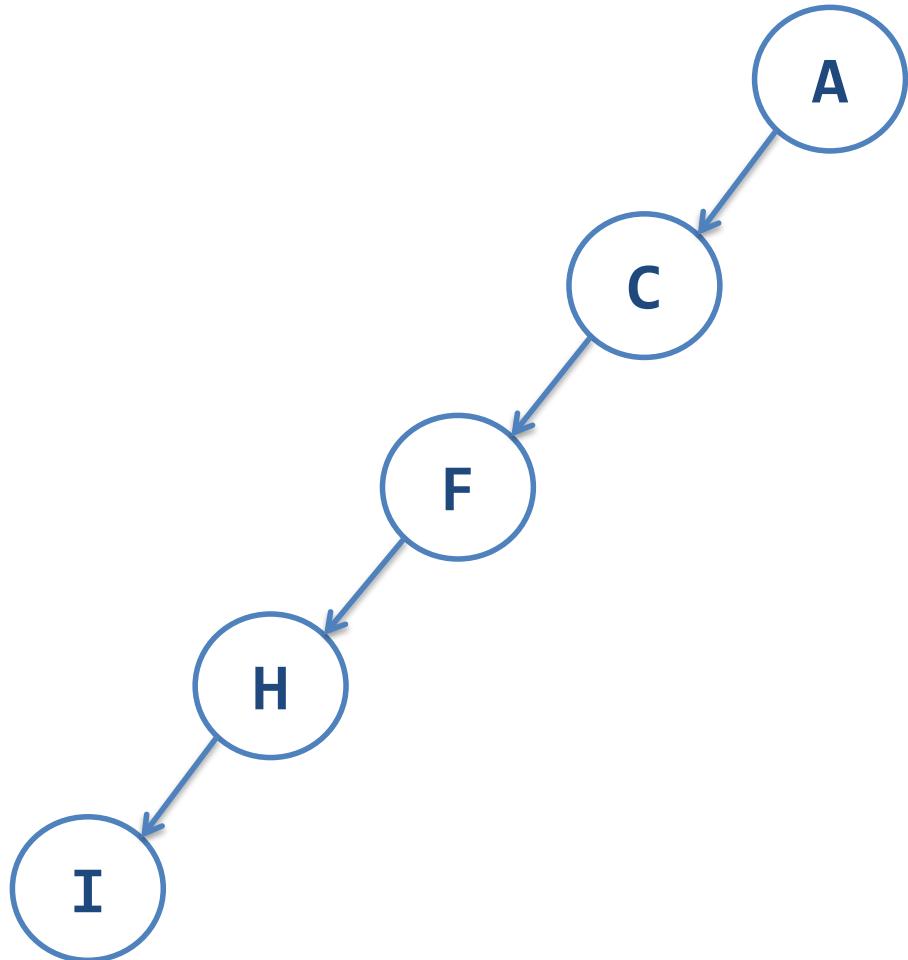
# Árvores Binárias: Sub-árvores

# Sub-árvore esquerda do nó A

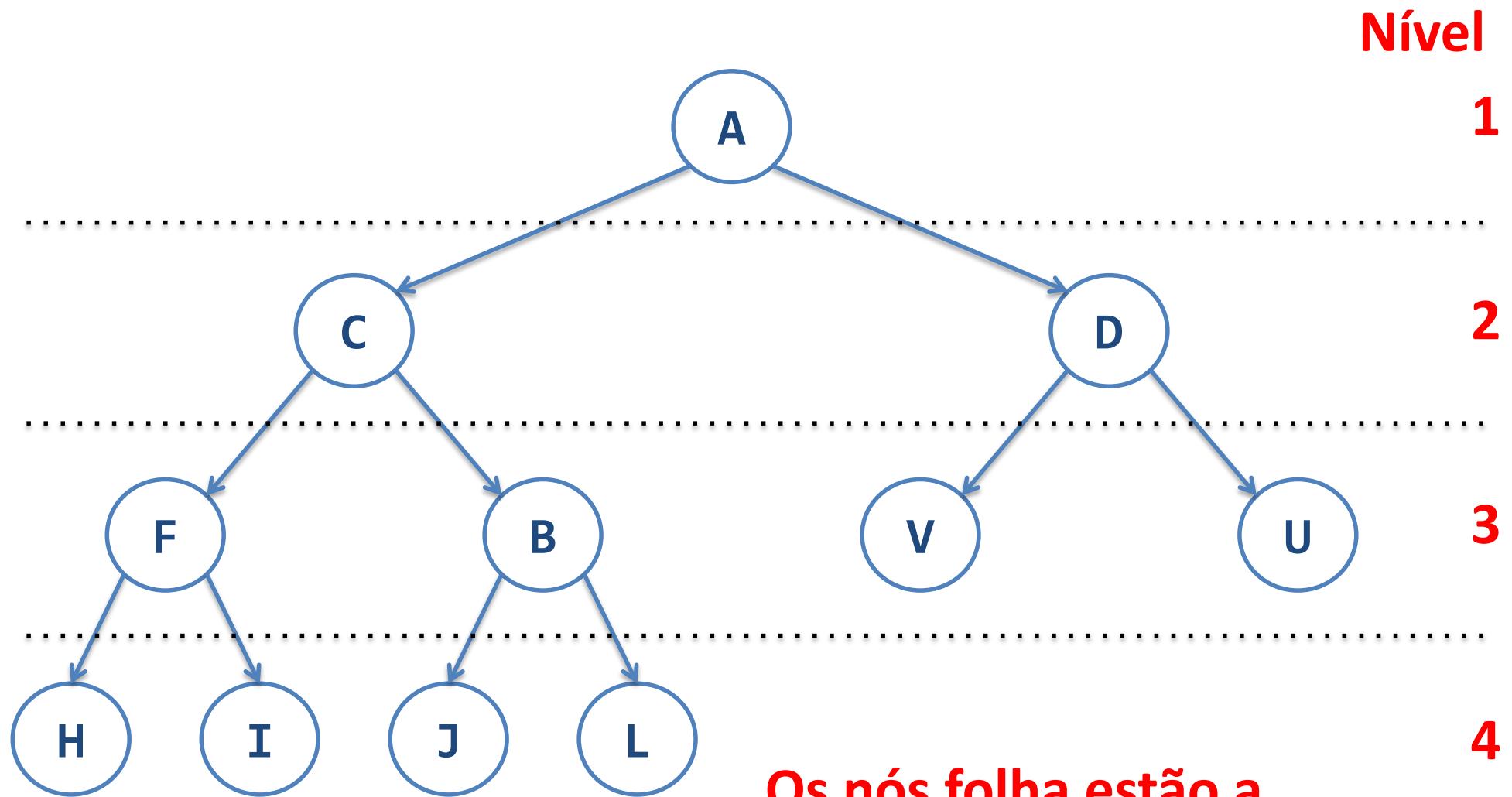


# Sub-árvore direita do nó A

# Árvores Binárias: Degeneradas



# Árvores Binárias: Completas



Os nós folha estão a  
um nível de diferença

# Árvores Binárias: Propriedades

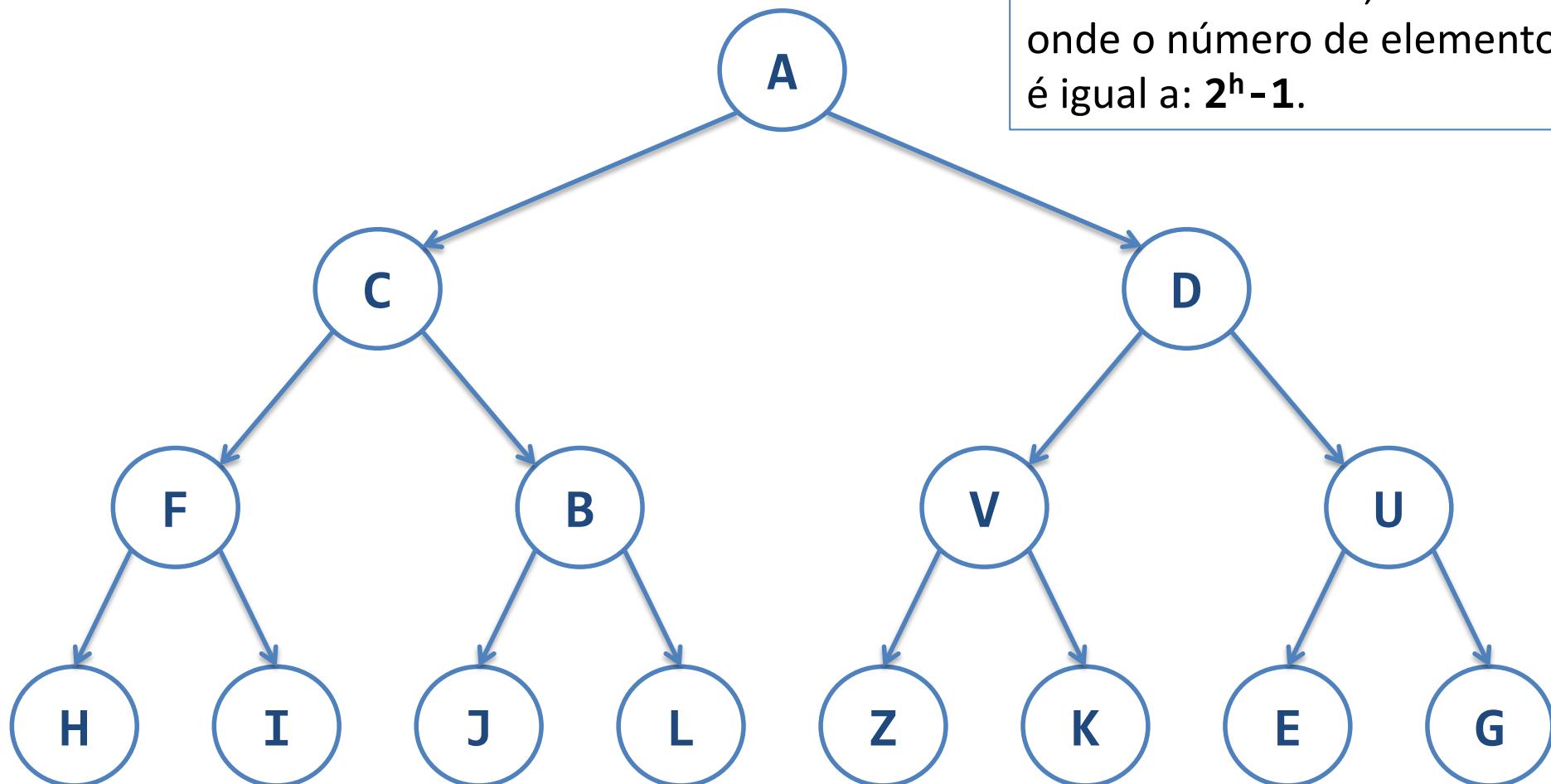
- Número máximo de nós no nível  $i$ :

$$2^{i-1}, i > 0$$

- Número máximo de nós de uma árvore binária de altura  $h$ :

$$\sum_{i=1}^h 2^{i-1} = 2^h - 1, h > 0$$

# Árvores Binárias: Plenas



Árvore de altura  $h$ , com  $h \geq 0$ , onde o número de elementos é igual a:  $2^h - 1$ .

# Árvores Binárias: Propriedades

- A altura de uma árvore binária plena com n elementos é:

$$n = 2^h - 1, h > 0$$

$$n + 1 = 2^h$$

$$\log_2(n + 1) = \log_2 2^h$$

$$\log_2(n + 1) = h \log_2 2$$

$$\therefore h = \log_2(n + 1)$$

# Árvores Binárias: Propriedades

- Se uma árvore plena possui  $n$  elementos na folha, o nível  $i$  das folhas será dado por:

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

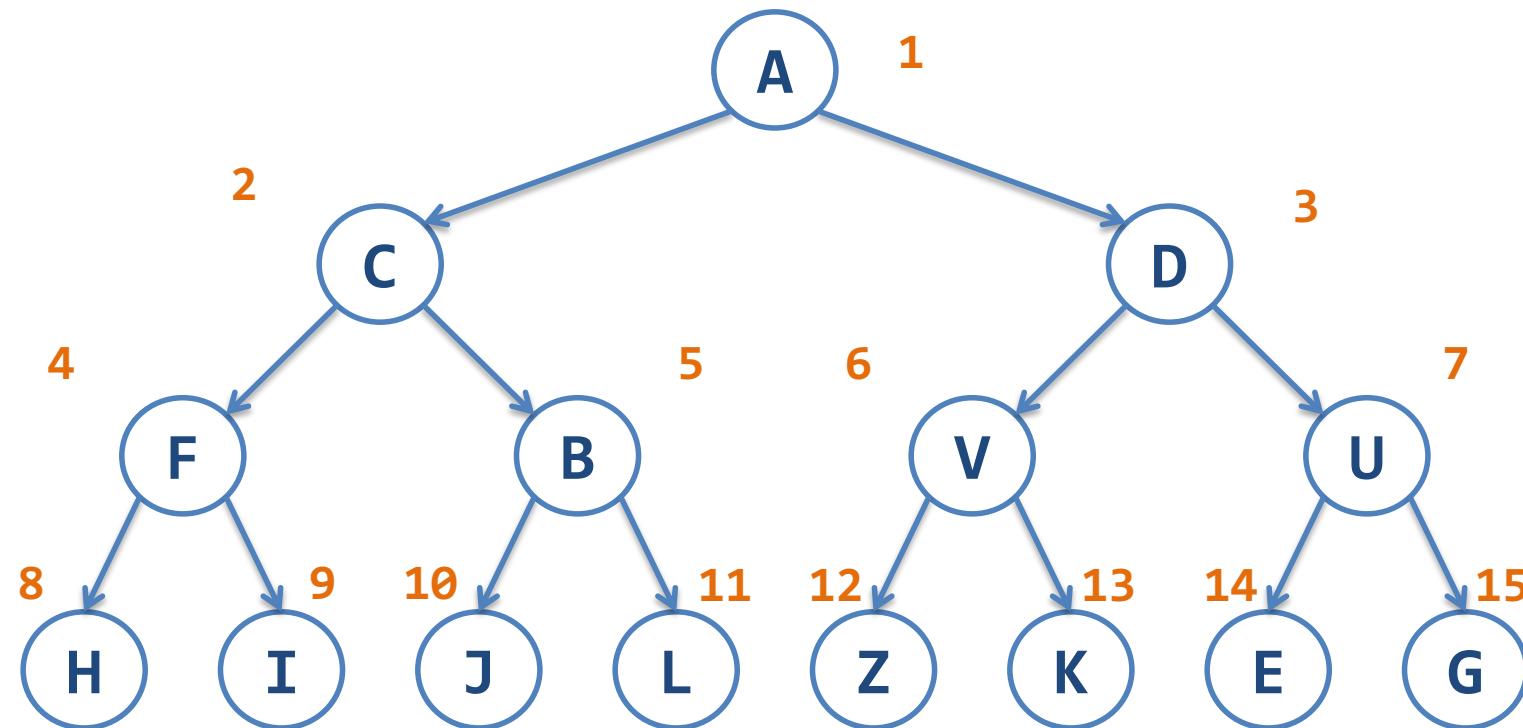
$$\log_2 2^{i-1} = \log_2 n$$

$$(i-1)\log_2 2 = \log_2 n$$

$$i = \log_2 n + 1$$

# Representação Sequencial de Árvores Binárias

# Representação Sequencial



Para uma árvore completa com  $n$  nós, qualquer nó com índice  $i$ ,  $1 \leq i \leq n$ , temos:

$$\text{parent}(i) = \lfloor i / 2 \rfloor, i \neq 1$$

$\text{left\_child}(i) = 2i$ , se  $2i \leq n$ ; se  $2i > n \Rightarrow i$  não possui filho à esquerda

$\text{right\_child}(i) = 2i + 1$ , se  $2i + 1 \leq n$ ; se  $2i + 1 > n \Rightarrow i$  não possui filho à direita

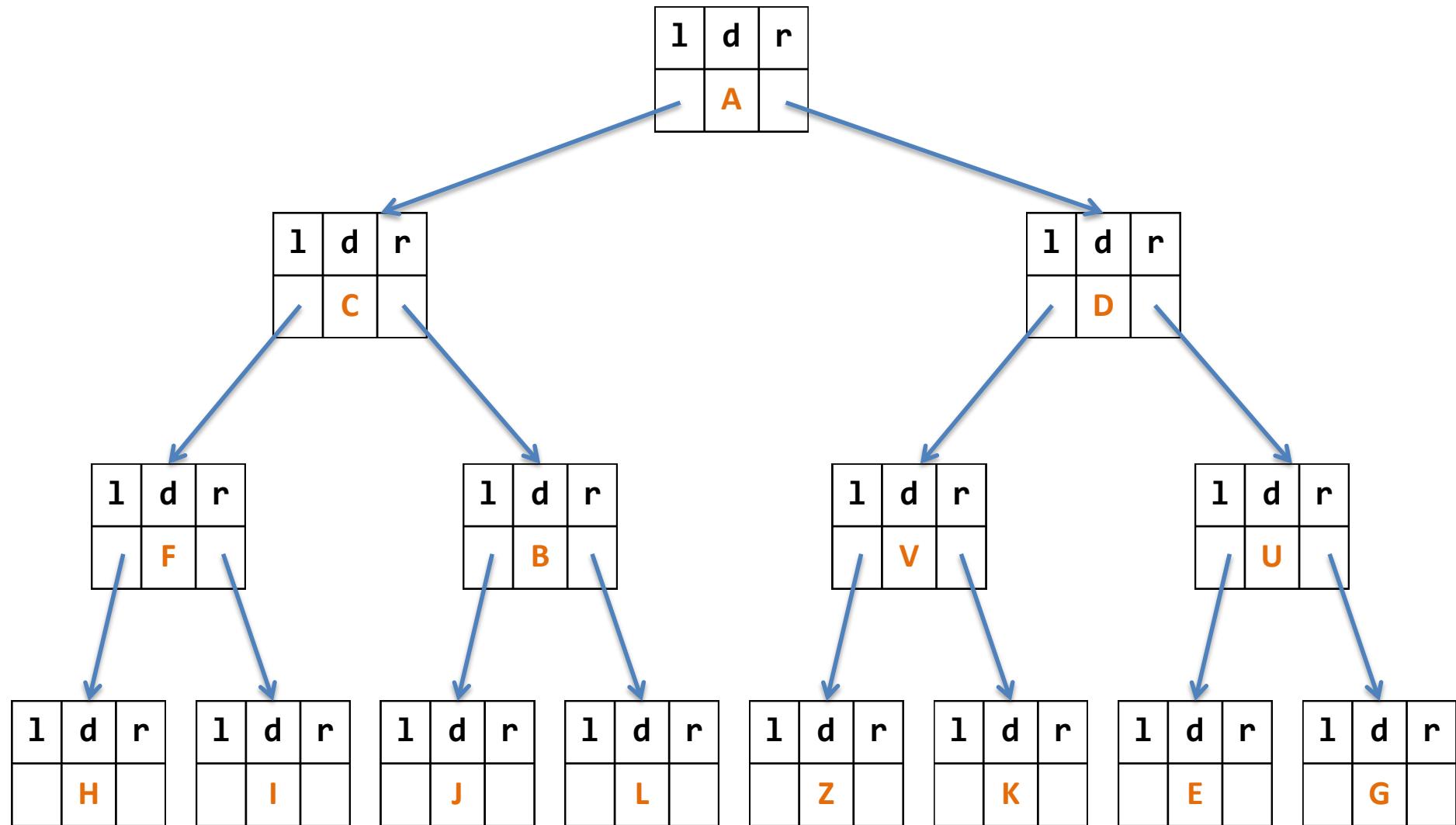
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	D	F	B	V	U	H	I	J	L	Z	K	E	G

O que é interessante sobre a representação  
sequencial de uma árvore binária por meio  
de um vetor?

Qual o problema com a representação  
sequencial de uma árvore binária por meio  
de um vetor?

# Representação de Árvores Binárias por meio de Ponteiros

# Representação por meio de Ponteiros



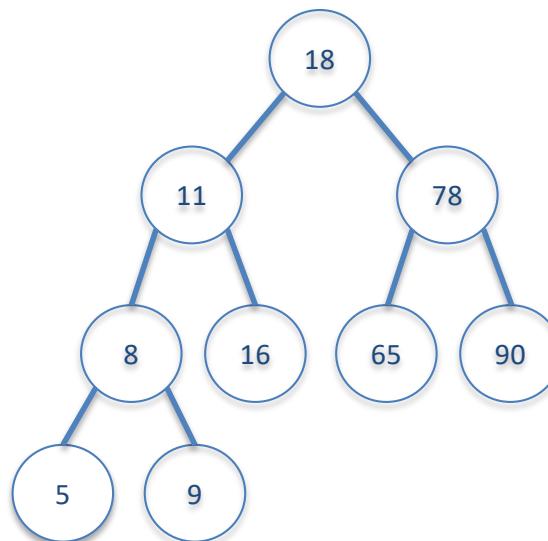
# Travessia em Árvores Binárias

# Travessia em Árvores Binárias

- Uma dos tipos de processamento que podemos realizar com uma árvore é aplicar uma certa computação em cada um de seus elementos.
- Isso significa visitar cada um dos nós da árvore, o que chamamos de **travessia da árvore**.
- Formas usuais de travessia:
  - Pré-ordem (Preorder)
  - Ordem-central (Inorder)
  - Pós-ordem (Postorder)

# Travessia Pré-ordem Recursiva

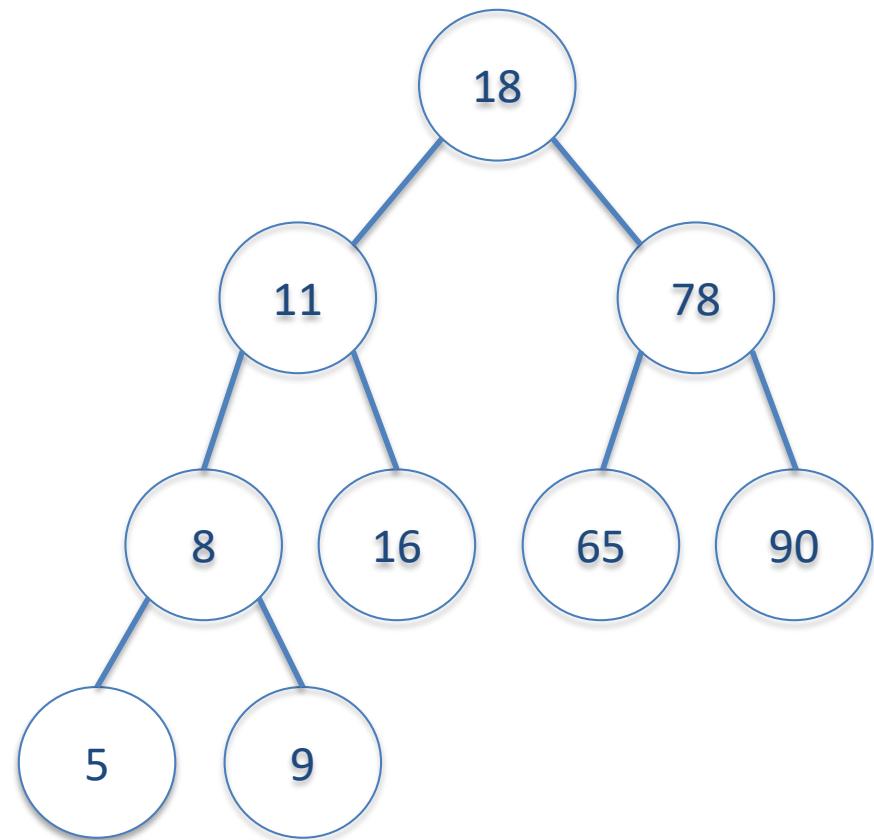
- Este tipo de travessia pode ser descrito de forma recursiva da seguinte maneira:
  - Aplica algum processamento na raiz.
  - Visita a sub-árvore esquerda em pré-ordem.
  - Visita a sub-árvore direita em pré-ordem.



# Travessia Pré-ordem Recursiva

```
void PreOrdem(node *n)
{
    if(n == nullptr)
        return;

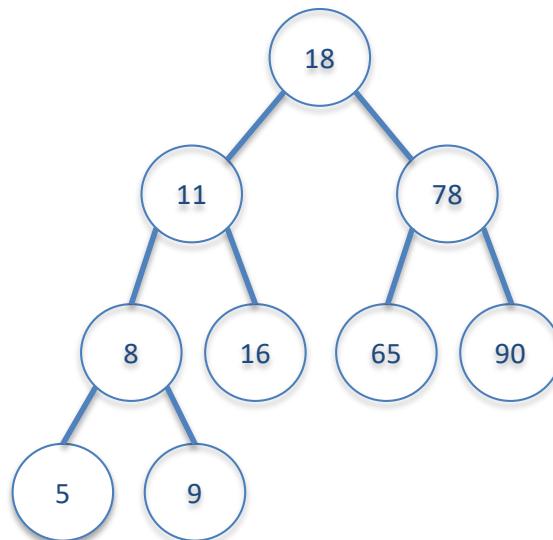
    std::cout << n->data;
    PreOrdem(n->left);
    PreOrdem(n->right);
}
```



Travessia: 18 -> 11 -> 8 -> 5 -> 9 -> 16 -> 78 -> 65 -> 90

# Travessia Pós-ordem Recursiva

- Este tipo de travessia pode ser descrito de forma recursiva da seguinte maneira:
  - Visita a sub-árvore esquerda em pós-ordem.
  - Visita a sub-árvore direita em pós-ordem.
  - Aplica algum processamento na raiz.



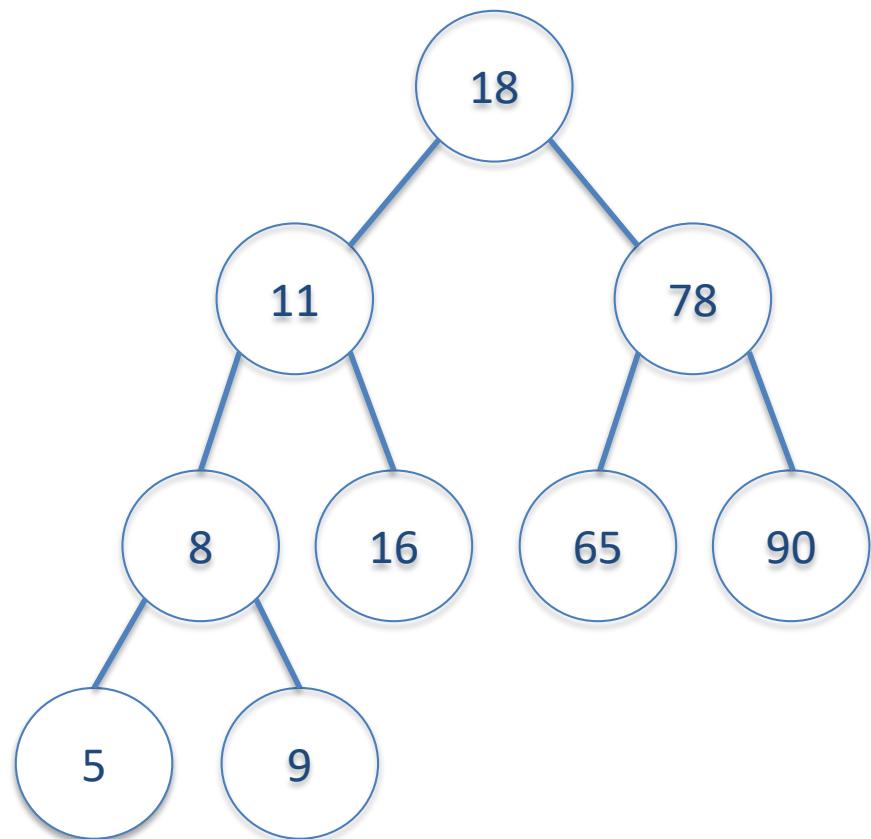
# Travessia Pós-ordem Recursiva

```
void PosOrdem(node *n)
{
    if(n == nullptr)
        return;

    PosOrdem(n->left);

    PosOrdem(n->right);

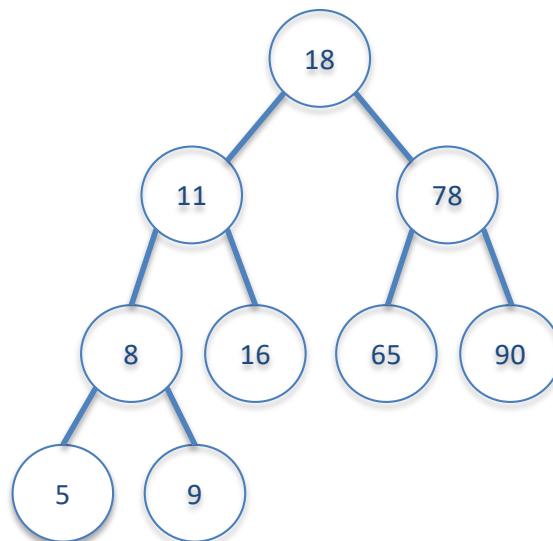
    std::cout << t->data;
}
```



Travessia: 5 -> 9 -> 8 -> 16 -> 11 -> 65 -> 90 -> 78 -> 18

# Caminhamento Central (*inorder*)

- Pode ser descrito de forma recursiva da seguinte maneira:
  - Visita a sub-árvore esquerda na ordem central.
  - Aplica algum processamento na raiz.
  - Visita a sub-árvore direita na ordem central.



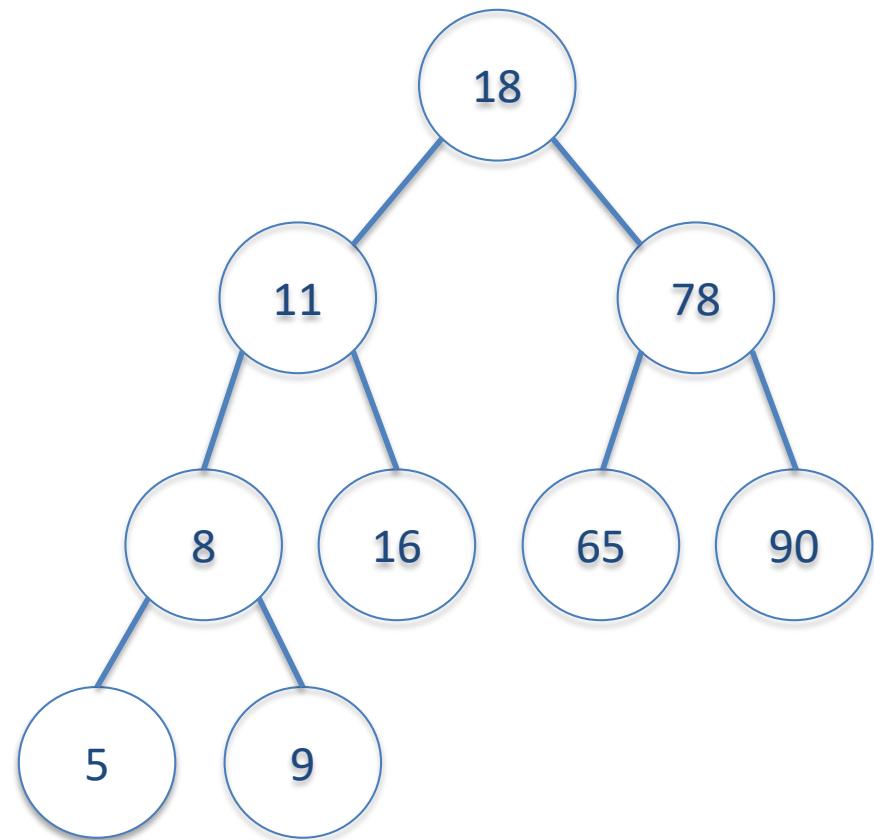
# Caminhamento Central (*inorder*)

```
void Central(node *n)
{
    if(n == nullptr)
        return;

    Central(n->left);

    std::cout << n->data;

    Central(n->right);
}
```



Travessia: 5 -> 8 -> 9 -> 11 -> 16 -> 18 -> 65 -> 78 -> 90

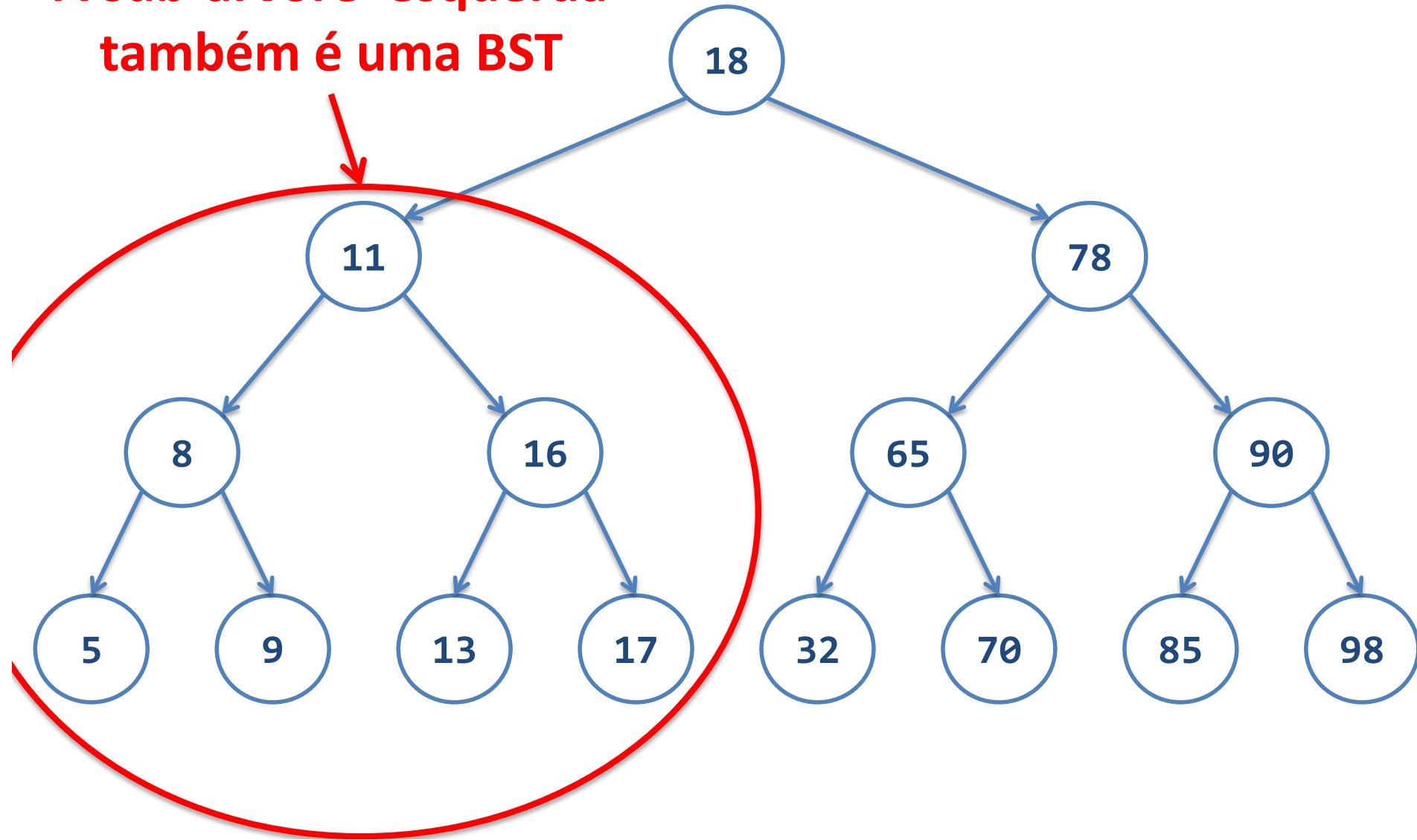
# Árvores Binárias de Pesquisa *(Binary Search Trees)*

# Árvores Binárias de Pesquisa

- Numa árvore binária de pesquisa:
  - Os elementos possuem chave única;
  - As chaves da sub-árvore esquerda são menores que a da sub-árvore direita;
  - As chaves da sub-árvore direita são maiores do que a da raiz;
  - As sub-árvores esquerda e direita também são árvores binárias de pesquisa.

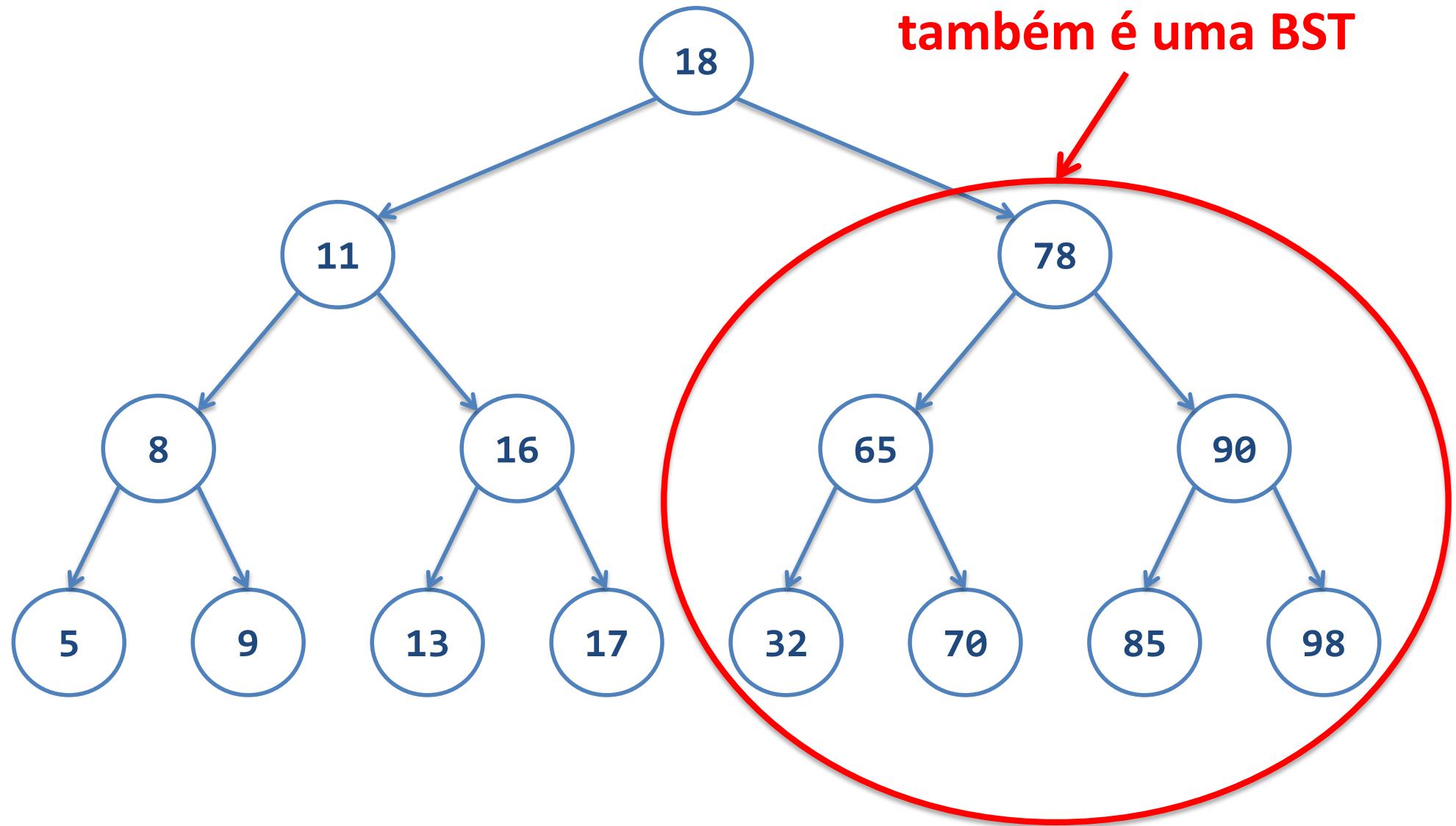
# Árvores Binárias de Pesquisa

A sub-árvore esquerda  
também é uma BST



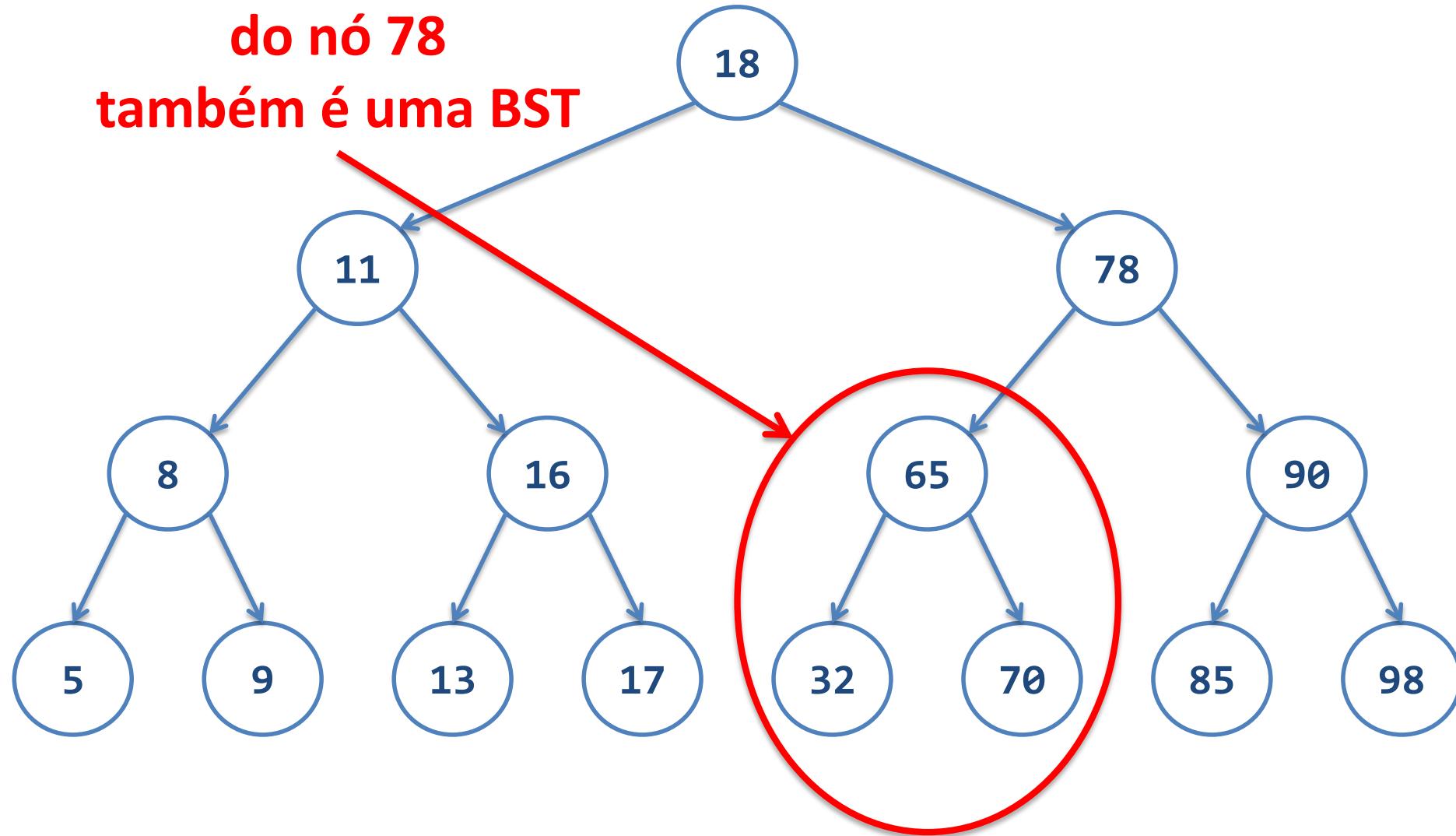
# Árvores Binárias de Pesquisa

A sub-árvore direita  
também é uma BST

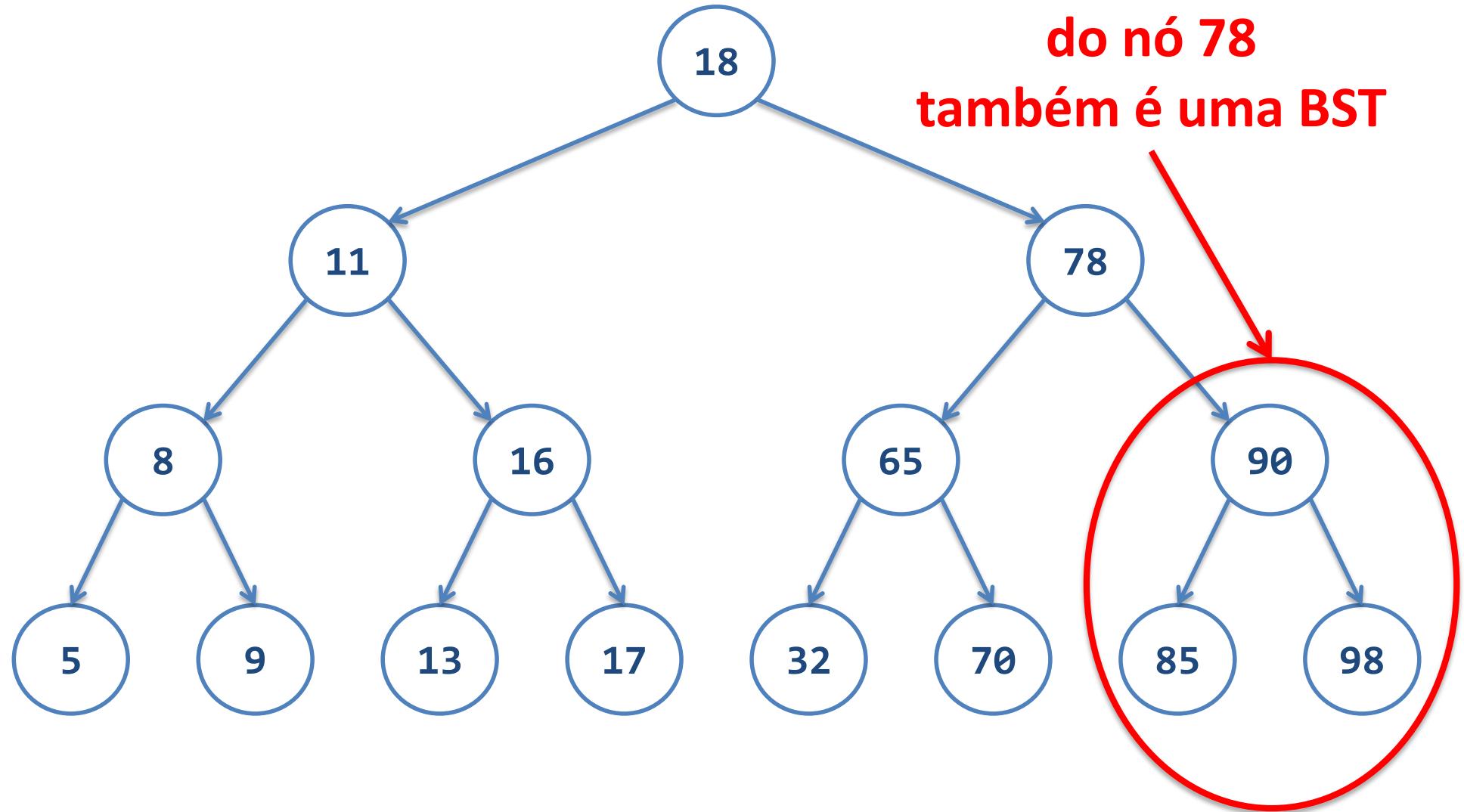


# Árvores Binárias de Pesquisa

A sub-árvore esquerda  
do nó 78  
também é uma BST



# Árvores Binárias de Pesquisa



A sub-árvore direita  
do nó 78  
também é uma BST

TAD bstree

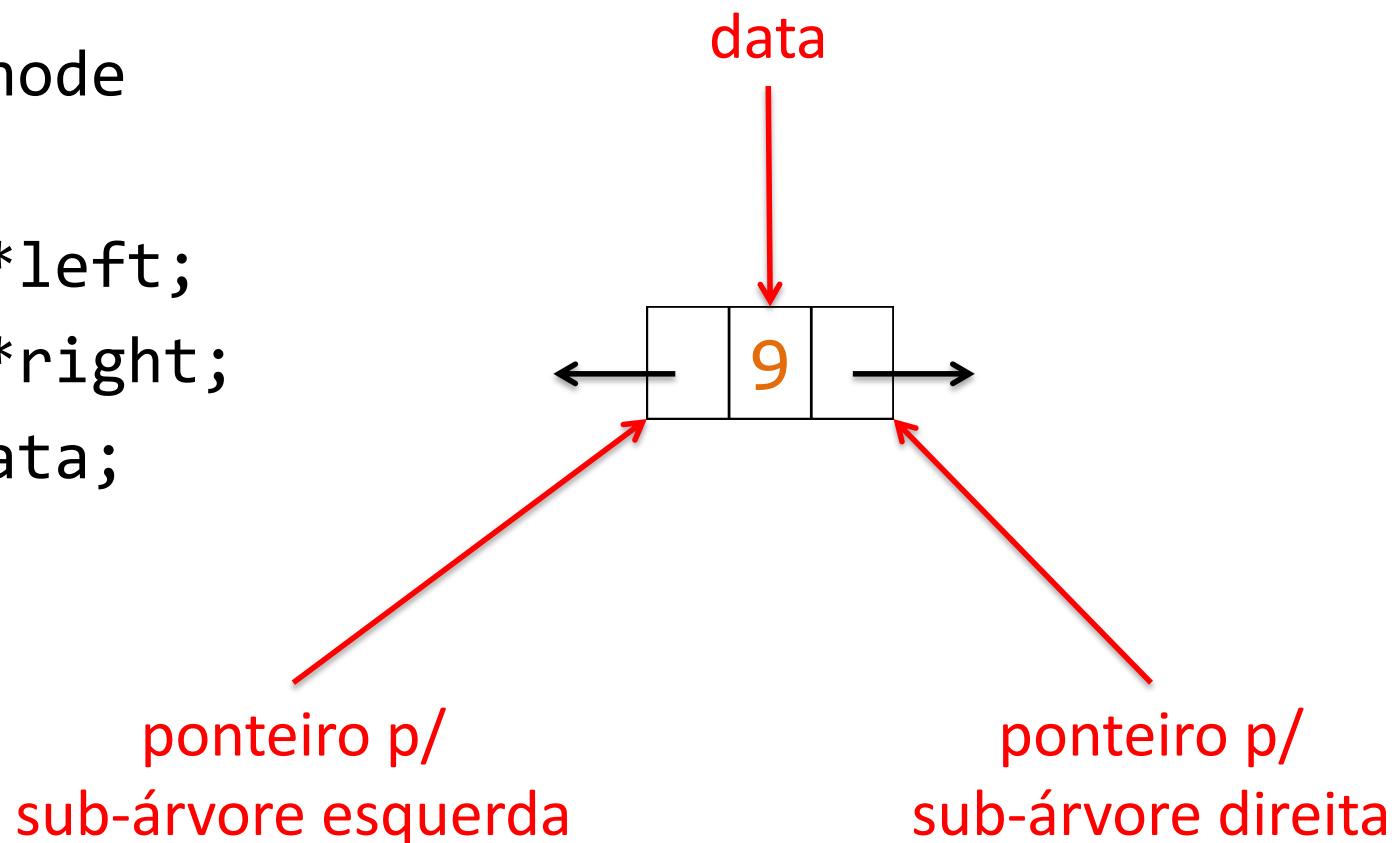
# TAD bstree: Operações Usuais

```
type bstree[item]
    new() → bstree
    insert(bstree, item) → bstree
    find(bstree, item) → bstree
    erase(bstree) → bstree
    empty(bstree) → boolean
```

# TAD bstree: Representação Interna

- Cada nó da árvore será representado pela seguinte estrutura:

```
struct node
{
    node *left;
    node *right;
    int data;
};
```

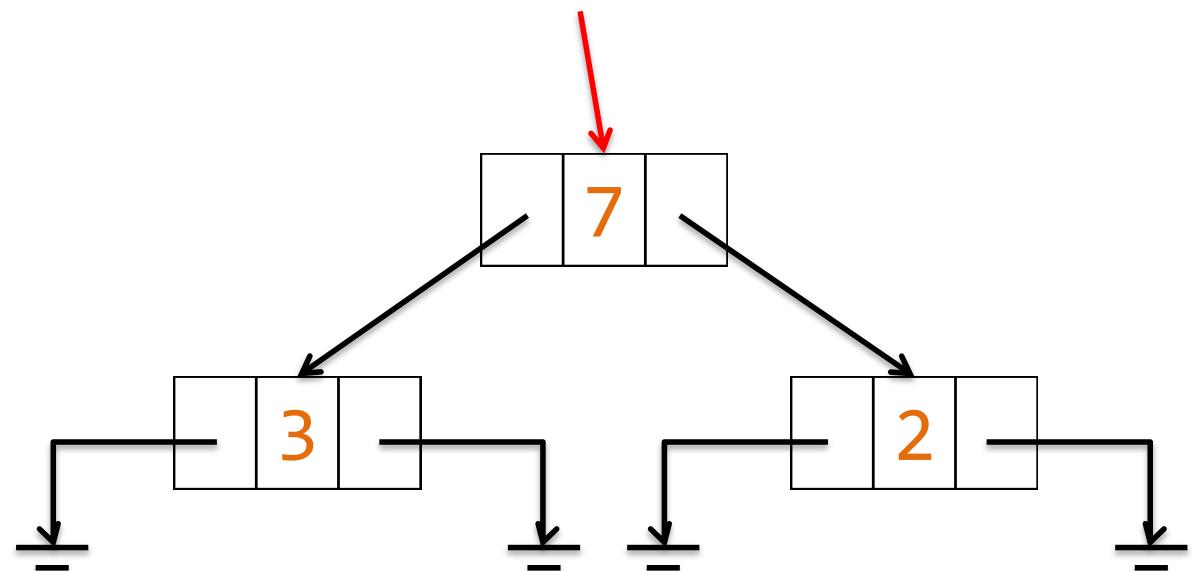


# TAD bstree: Representação Interna

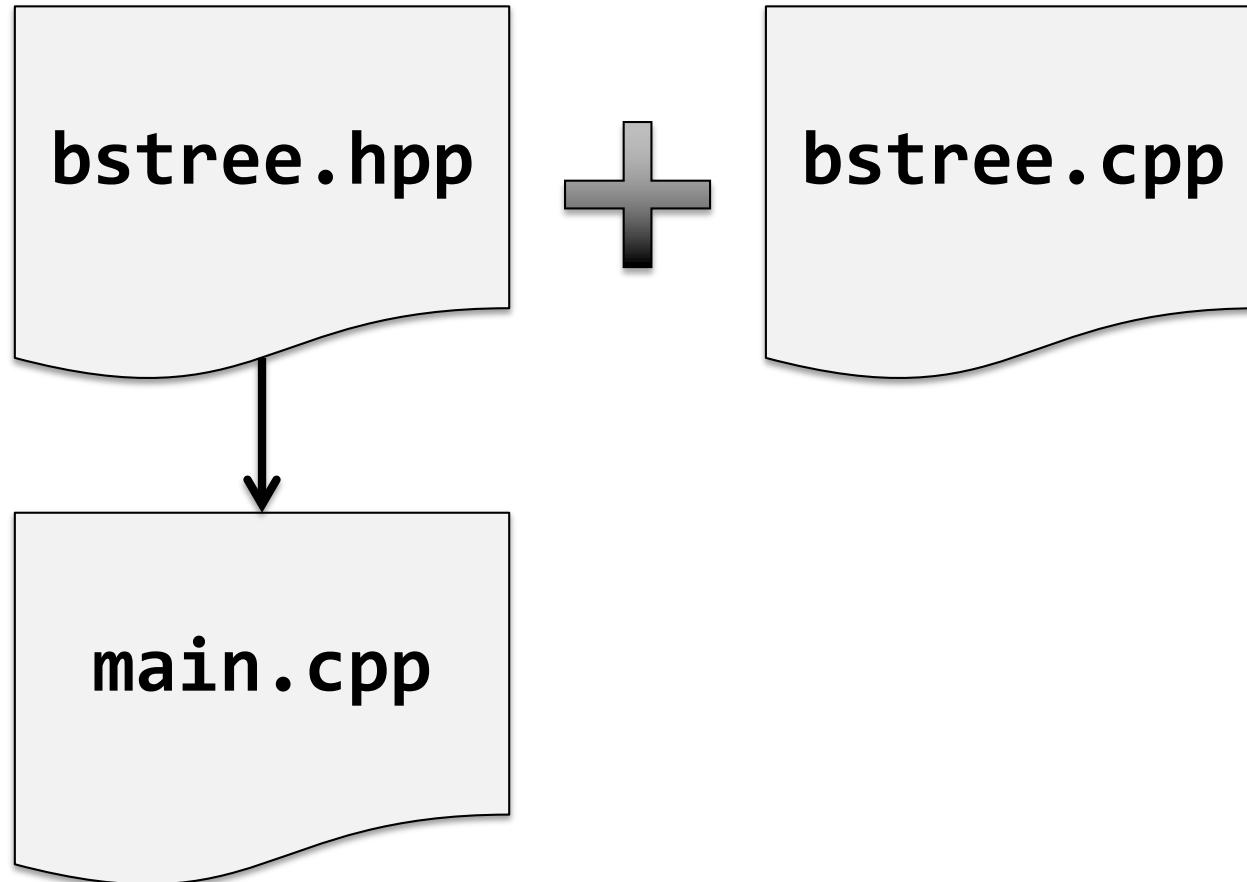
- Por sua vez, a árvore será representada por uma classe com um único membro de dados privado:

```
class bstree
{
    ...
private:
    node *root_;
};
```

**root\_**: ponteiro para  
a raiz da árvore



# TAD bstree: Organização Código



Para construir o programa com g++:

```
$ g++ -o teste_tree bstree.cpp main.cpp -std=c++11
```

# TAD bstree: Definição

```
class bstree
{
public:
    struct node
    {
        public:
            int data;
    private:
        node *left;
        node *right;
        friend class bstree;
    };
    ...
};
```

Estrutura aninhada

Os membros de dados das sub-árvores são privados

Permite que funções membros da classe bstree possam acessar os membros de dados privados de um node

# TAD bstree: Definição

```
class bstree
{
public:
    ...
    bstree();
    ~bstree();
    ...
};
```

Construtor

Destrutor

# TAD bstree: Definição

```
class bstree
{
public:
    ...
    node* find(int v) const;
    node* insert(int v);
    void erase(int v);
    ...
};
```

# TAD bstree: Definição

```
class bstree  
{
```

```
    ...
```

```
private:
```

```
    bstree(bstree&);
```

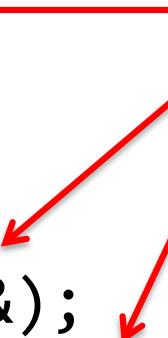
```
    bstree& operator=(bstree&);
```

```
private:
```

```
    node *root_;
```

```
};
```

Tornando estes dois operadores privados, sem fornecer uma implementação, fará com que os objetos bstree não possam ser copiáveis



# TAD bstree: Implementação

- **find:** procura pelo elemento de chave v ou, caso não seja encontrado, retorna nulo.

```
node* bstree::find(int v) const
{
    return find(v, root_);
}
```

```
node* bstree::find(int v, node *n) const
{
    if(n == nullptr)
        return nullptr;

    if(n->data == v)
        return n;

    return (v < n->data) ? find(n->left, v) : find(n->right, v);
}
```

Busca de Forma Recursiva

# TAD bstree: implementação

- **find:** procura pelo elemento de chave v ou, caso não seja encontrado, retorna nulo.

```
node* bstree::find(int v) const
{
    node *n = root_;
    while(n)
    {
        if(n->data == v)
            break;
        n = (v < n->data) ? n->left : n->right;
    }
    return n;
}
```

Busca de forma Iterativa

# Árvores Binárias de Pesquisa: Busca

- Qual a complexidade do algoritmo de busca em uma árvore binária de pesquisa?
- Se a árvore possui altura  $h$ , ela será pesquisada em tempo  $O(h)$ .
- Em uma árvore completamente balanceada:

$$h = \log_2(n + 1) \Rightarrow O(\log_2 n)$$

# TAD bstree: Implementação

- **insert:** insere na árvore um novo elemento com chave v.

```
node* bstree::insert(int v)
{
    node* n = root_;

    node* parent = nullptr;

    while(n)
    {
        if(n->data == v)
            throw std::logic_error("duplicated key!");

        parent = n;

        n = (v < n->data) ? n->left : n->right;
    }

    node* nn = new node(v, nullptr, nullptr);

    if(parent == nullptr)
        root_ = nn;
    else if(v < parent->data)
        parent->left = nn;
    else
        parent->right = nn;

    return nn;
}
```

```
node* bstree::insert(int v)
{
    node *n = root_; ← Começando pela raiz
    node *parent = nullptr; ← Guardar onde a busca parou

    while(n) ← Procurar onde inserir o novo
    {           elemento
        if(n->data == v)
            throw std::logic_error("duplicated key!");

        parent = n; ← Nó anterior a próxima busca

        n = (v < n->data) ? n->left : n->right;
    }

    node *nn = new node(v, nullptr, nullptr); ← Cria um novo elemento

    if(parent == nullptr) ← A árvore estava vazia?
        root_ = nn;
    else if(v < parent->data) ← Em qual sub-árvore faremos a
        parent->left = nn;           inserção?
    else
        parent->right = nn;

    return nn;
}
```

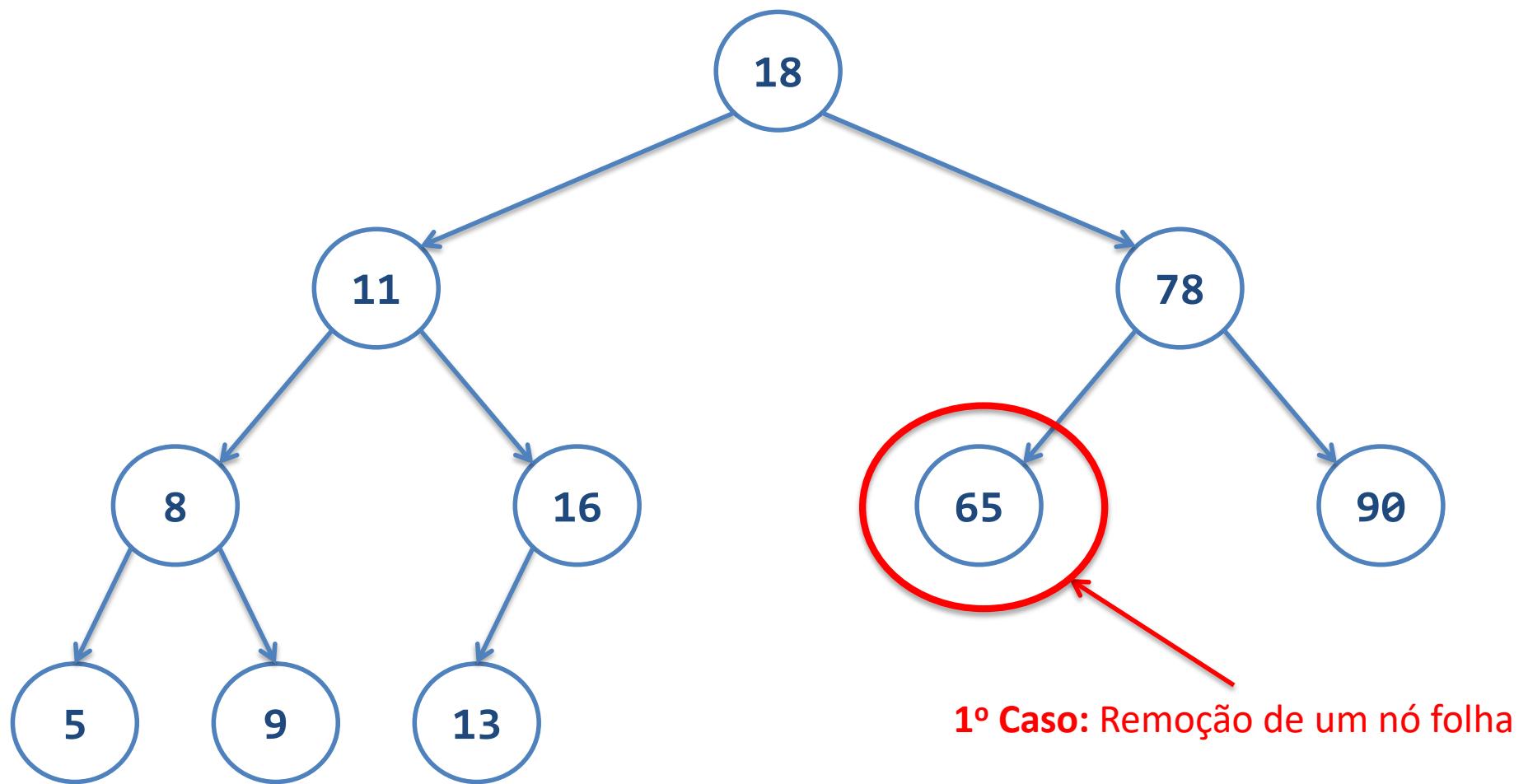
# Árvores Binárias de Pesquisa: Inserção

- Qual a complexidade do algoritmo de inserção em uma árvore binária de pesquisa?
- Se a árvore possui altura  $h$ , ela será pesquisada em tempo  $O(h)$ , até achar o local de realizar a inserção.
- Em uma árvore completamente balanceada:

$$h = \log_2(n + 1) \Rightarrow O(\log_2 n)$$

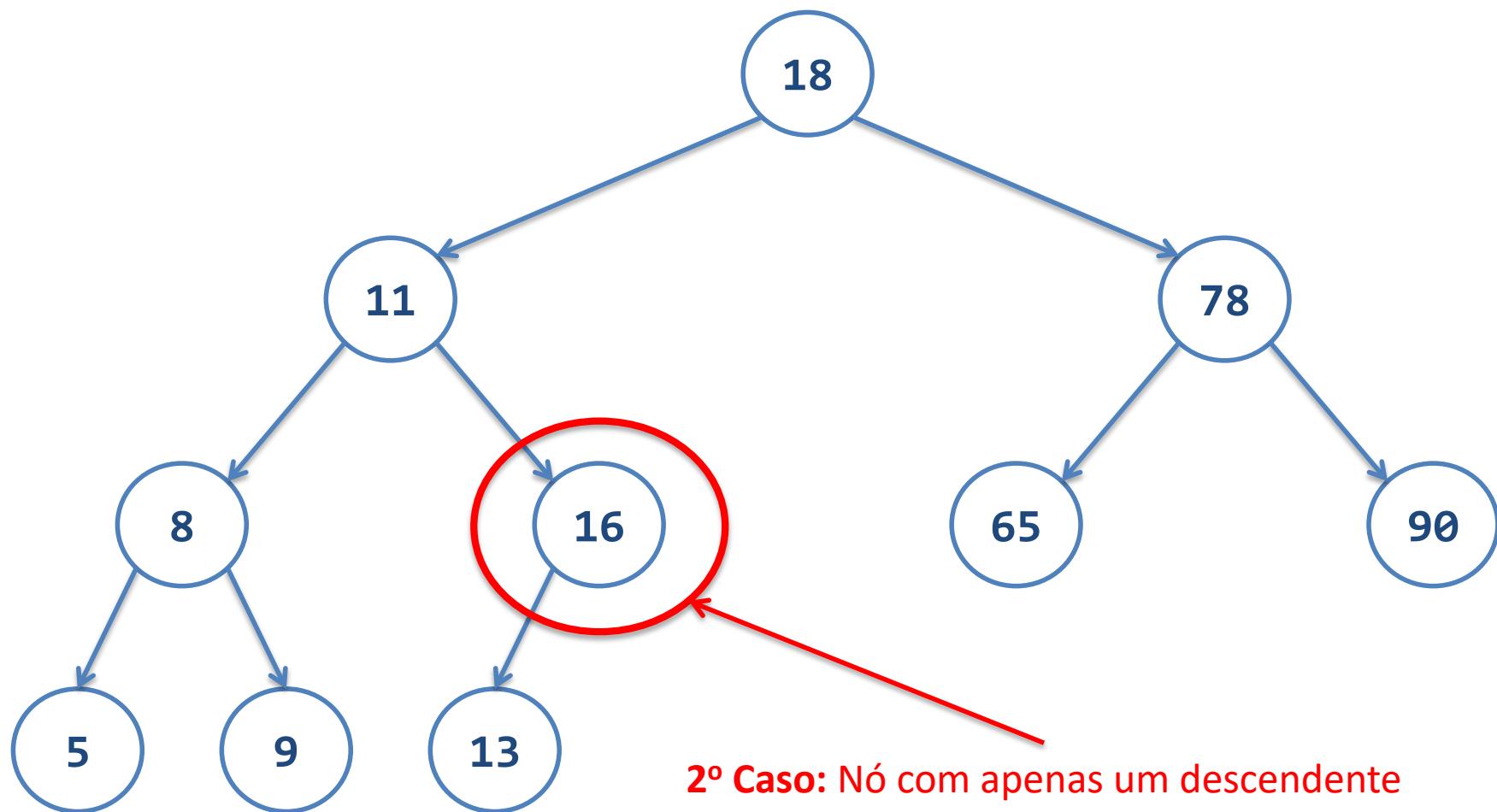
# TAD bstree: Implementação

- **erase**: remove da árvore o elemento de chave v.



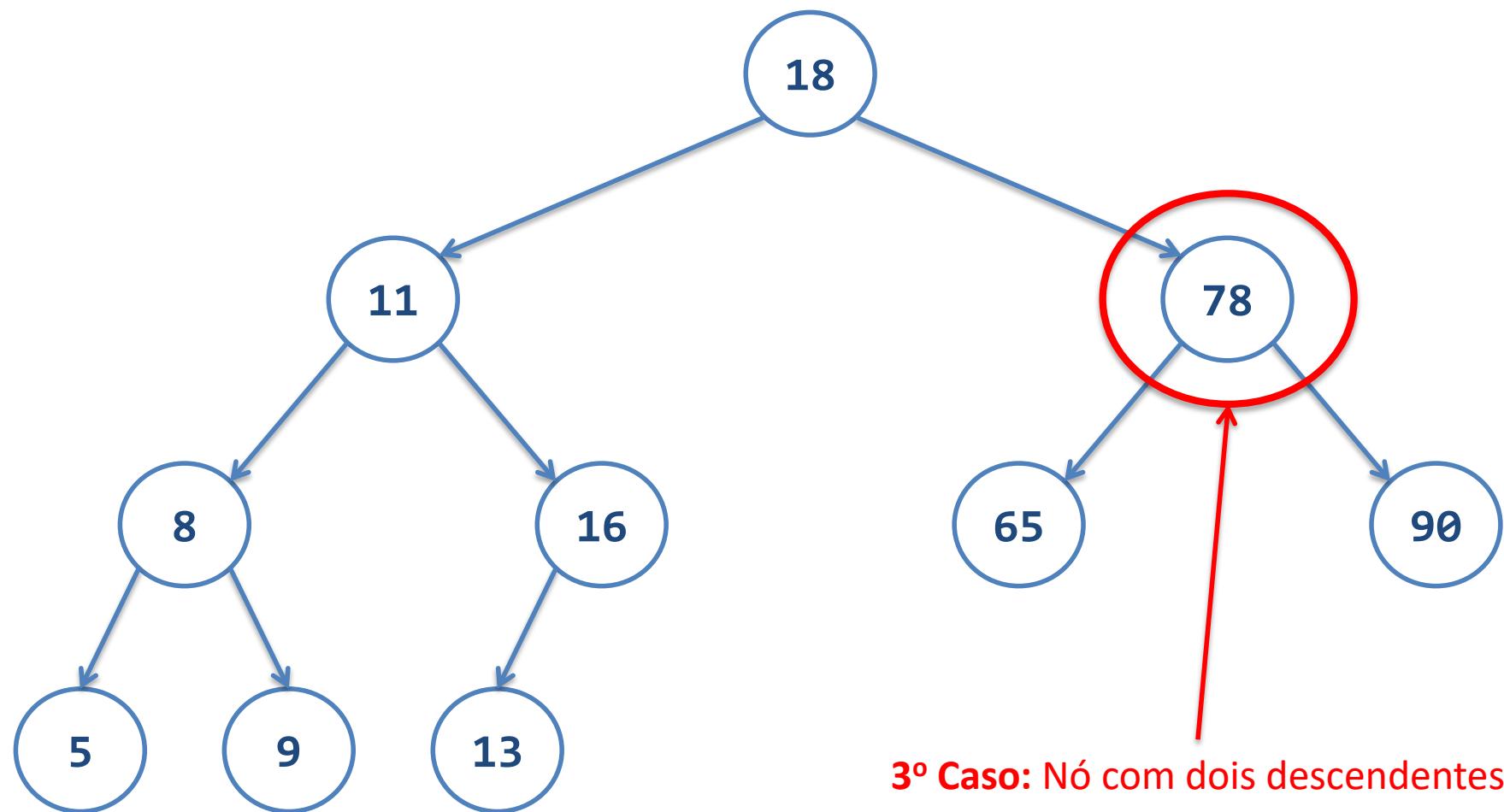
# TAD bstree: Implementação

- **erase**: remove da árvore o elemento de chave v.



# TAD bstree: Implementação

- **erase**: remove da árvore o elemento de chave v.



3º Caso: Nó com dois descendentes

# Árvores Binárias de Pesquisa: Remoção

- Qual a complexidade do algoritmo de remoção em uma árvore binária de pesquisa?
- Se a árvore possui altura  $h$ , ela será pesquisada em tempo  $O(h)$ .
- Encontrar o elemento para substituição pode ser feito também em no máximo  $O(h)$ .
- Em uma árvore completamente balanceada:

$$h = \log_2(n + 1) \Rightarrow O(\log_2 n)$$

# Standard Template Library (STL)

set, multiset, map, multimap,  
priority\_queue

# Considerações Finais

# Considerações Finais

- As árvores são uma das estruturas mais utilizadas na computação.
- Os Sistemas de Bancos de Dados Relacionais utilizam estruturas de árvore para fornecer métodos de acesso ou índices que ajudam a acelerar o processamento das consultas.
- As aplicações Geoespaciais utilizam árvores multidimensionais para lidar com os dados espaciais e suas representações geométricas: pontos, linhas e polígonos.

# Considerações Finais

- Os compiladores utilizam as árvores para representar a estrutura sintática dos programas.
- As árvores são importantes para para construção de estruturas de dados que ajudem a suportar as operações de *Merge* e *Find* em conjuntos.
- Análise de circuitos elétricos.

# Considerações Finais

- Árvores em Linguagens de Programação:
  - Em C++, esta estrutura é o pilar fundamental dos seguintes containers da STL:
    - `std::set`, `std::multiset`, `std::map`, `std::multimap`
  - A notar que esses contêineres da STL utilizam um tipo especial de árvore binária de pesquisa, que veremos mais adiante:
    - Árvores Binárias Balanceadas, em especial, as *red-black trees*.

# Considerações Finais

- A inserção de forma ordenada de  $n$  chaves em uma árvore binária de pesquisa produz uma árvore de altura  $O(n)$ .
- As árvores binárias de pesquisa com altura  $O(\log_2 n)$  são conhecidas como árvores balanceadas.
- As estruturas de dados mais conhecidas para suporte a árvores balanceadas são\*:
  - AVL
  - Red-Black
  - Splay-trees (zig-zag)
  - Árvores 2-3
  - Árvores-B

\*Iremos estudar algumas dessas árvores balanceadas mais adiante no curso.

# Árvores: Considerações Finais

- O poder real de uma estrutura dinâmica como a árvore pode ser melhor percebido quando temos um conjunto de dados que não é estático.
- As operações de travessia em árvore – pré-ordem, pós-ordem e caminhamento central, podem ser implementadas de forma iterativa, utilizando-se filas e pilhas, dependendo do tipo de travessia.

# Considerações Finais

- Algumas outras operações interessantes sobre árvores binárias:
  - Sucessor e Antecessor de um nó
  - Associação de rank aos elementos: 5º menor valor!
- Temos diversas formas para representar árvores.
- Mais adiante no curso, iremos aprofundar nas estruturas de dados de árvores: **árvores balanceadas**.
- A seguir veremos um tipo de árvore especial usada para construção de filas de prioridade.

Pittsburgh, Pennsylvania 15213

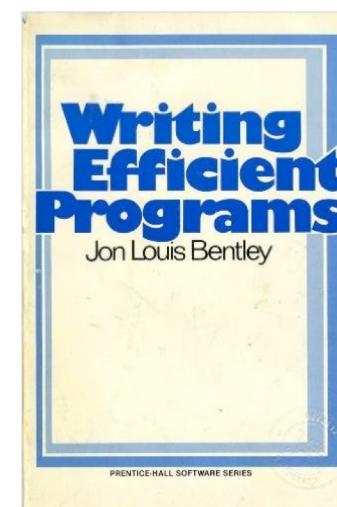
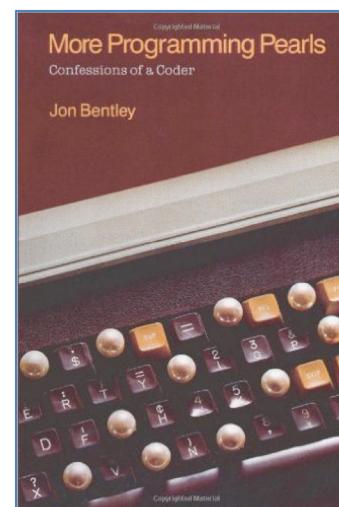
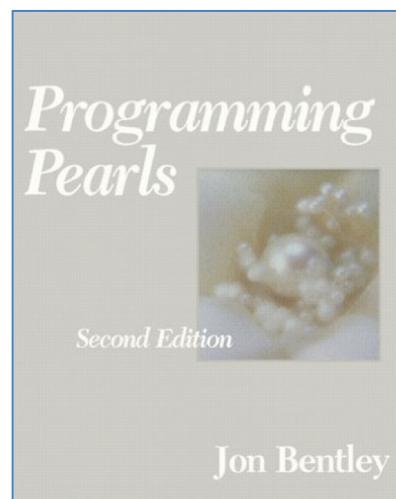
27 April 1981



Jon Louis Bentley

**Abstract ..** The most important step in making a software system efficient is the proper selection of data structures and algorithms; many papers and textbooks have been devoted to these topics. Most discussions, however, neglect another important activity: that of writing machine-independent efficient code. This paper examines a set of techniques for accomplishing that step. We will examine those techniques both in an abstract setting and in their application to a real program, where they led to a speedup of a factor of over six. Because these techniques should be employed rarely, an important part of this paper is describing exactly when one should (and should not!) use them.

Copyright (c) 1981, Jon Louis Bentley.



# Referências Bibliográficas

# Referências Bibliográficas

- Ziviani, N. **Projeto de Algoritmos: com implementações em PASCAL e C.** 2<sup>a</sup> Edição. Editora Thomson, 2005. 552 p.
- Sedgewick, R. **Algorithms.** 2<sup>a</sup> Edição. Addison-Wesley, 1988. 657 p.
- Cormen, T. H.; Lieserson, C. E.; Rivest, R. L.; Stein, C. **Introduction to Algorithms.** 3<sup>a</sup> Edição. Mit Press, 2009. 1312 p.
- Horowitz, E.; Sahni, S.; Rajasekaran, S. **Computer Algorithms.** 1<sup>a</sup> Edição. Computer Science Press, 1997. 769 p.

