



Algoritmos e Estruturas de Dados

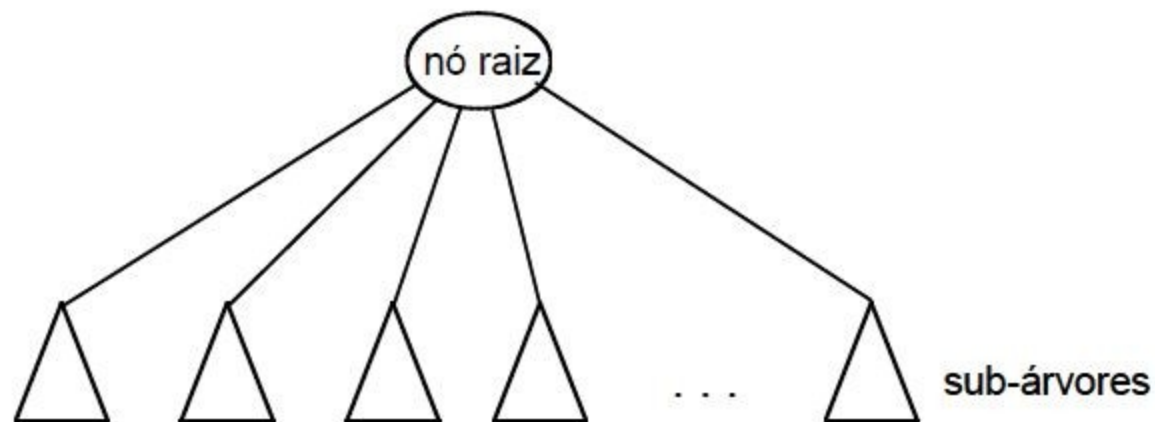
Estrutura de dados: Árvore – Capítulo 8
2019/20

Estrutura de dados: Árvore



Árvore é um conjunto finito de nós, onde se encontram os elementos. Esse conjunto:

- Ou é vazio;
- Ou consiste num nó especial (denominado raiz) e num número finito de sub-árvores.

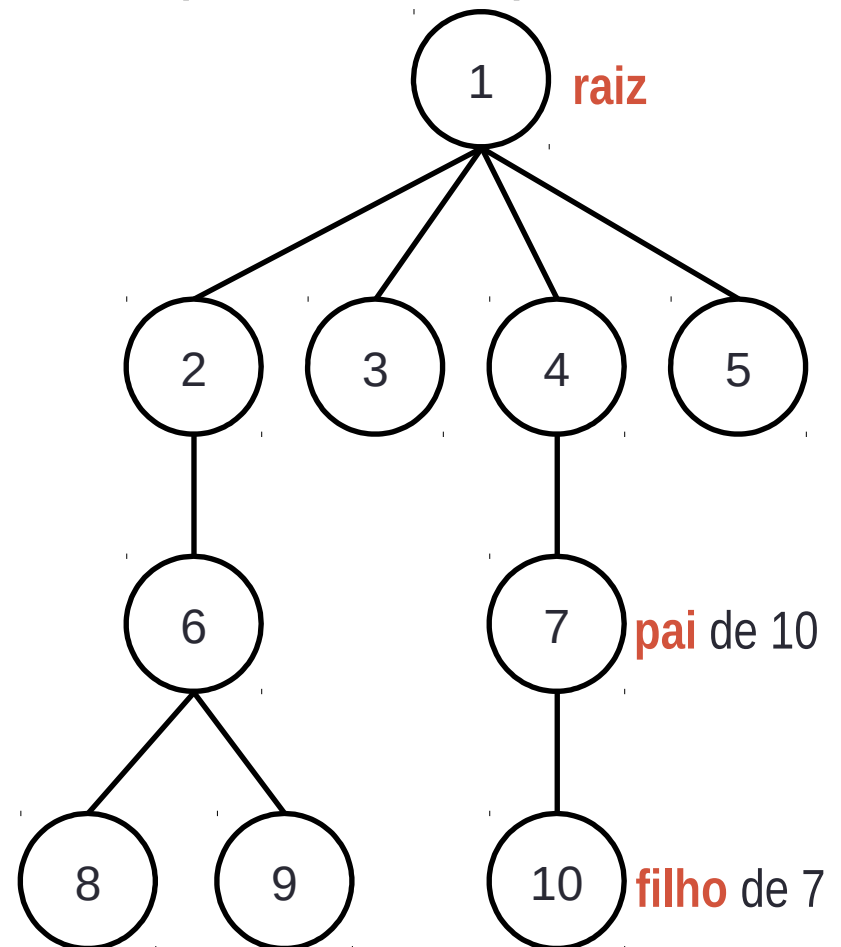


Estrutura de dados: Árvore



Na **Árvore**:

- Cada nó, excepto o nó raiz, tem um nó que é o seu pai (relação de **filho-pai**);

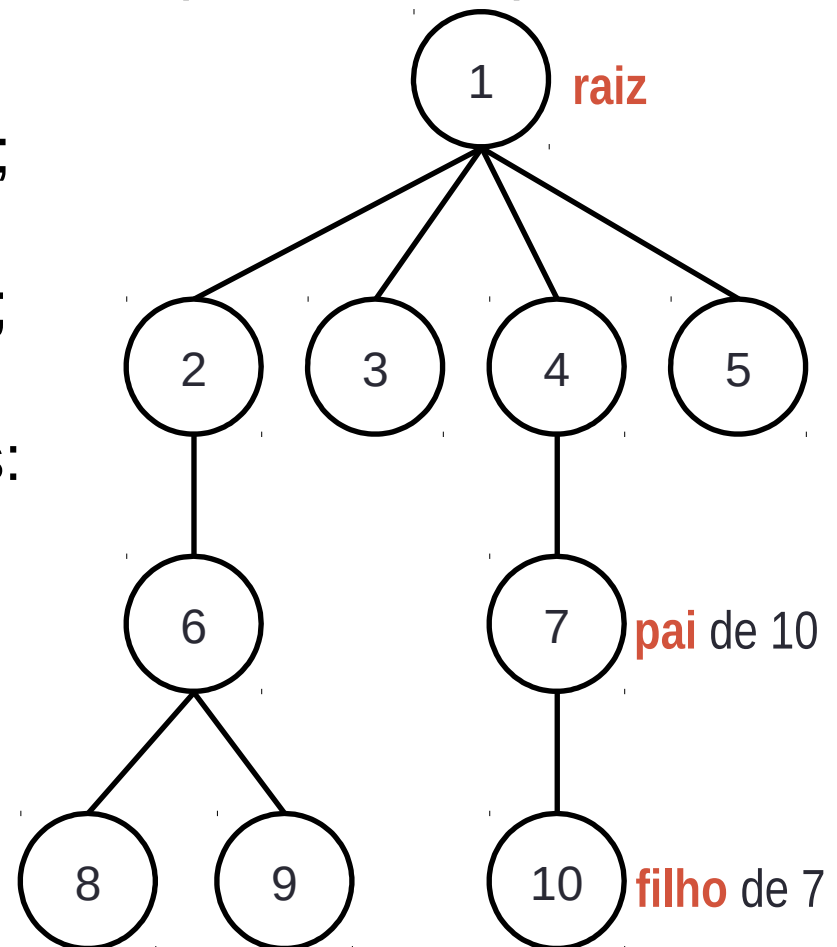


Estrutura de dados: Árvore



Na Árvore:

- Cada nó, excepto o nó raiz, tem um nó que é o seu pai (relação de **filho-pai**);
- Cada nó pode ter 0 ou mais filhos; Nó com zero filhos é denominado de **folha** (exemplos: 8, 9, 3, 10, 5); Nó com um ou mais filhos é denominado **nó interno** (exemplos: 1, 2, 6, 4, 7);

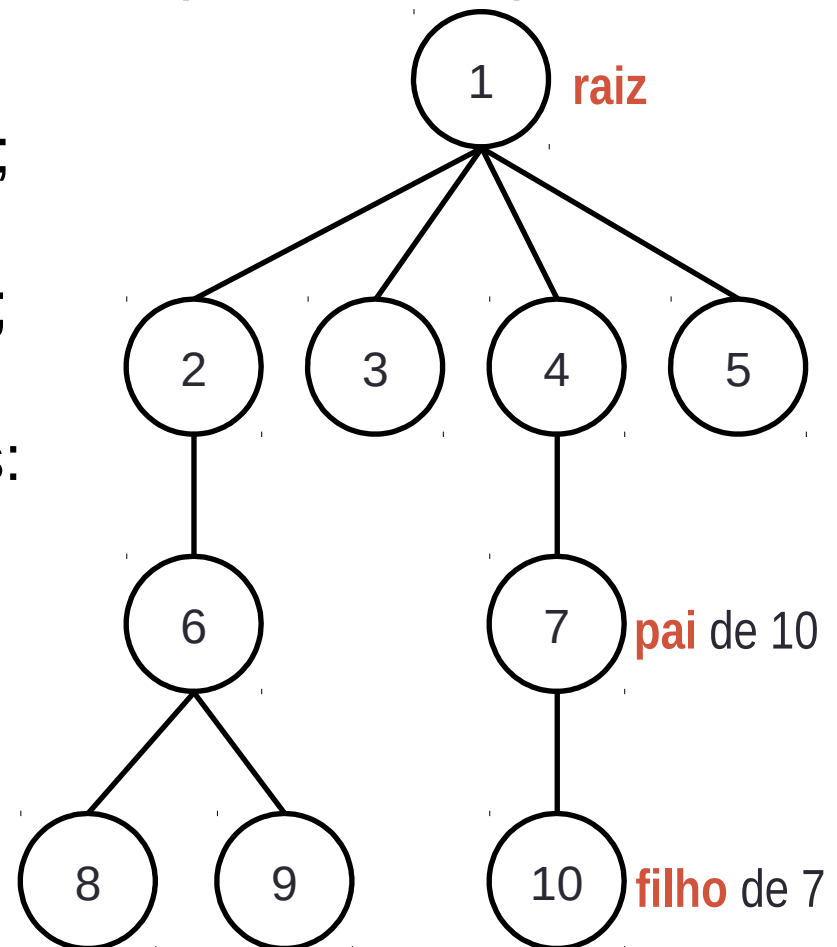


Estrutura de dados: Árvore



Na Árvore:

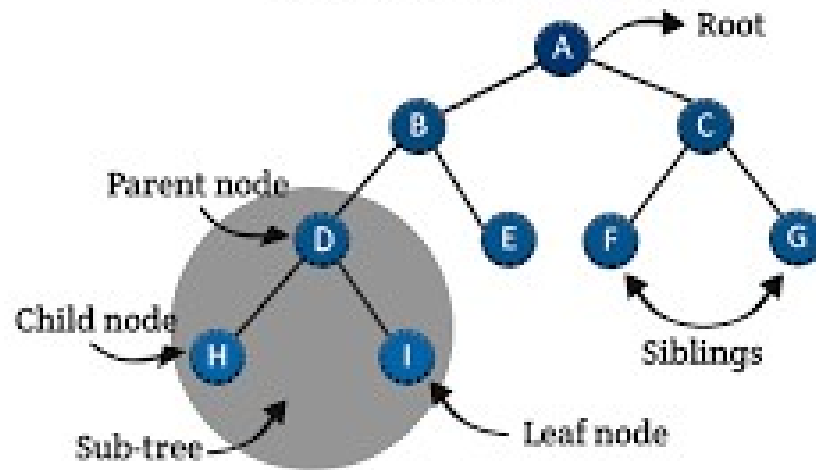
- Cada nó, excepto o nó raiz, tem um nó que é o seu pai (relação de **filho-pai**);
- Cada nó pode ter 0 ou mais filhos; Nó com zero filhos é denominado de **folha** (exemplos: 8, 9, 3, 10, 5); Nó com um ou mais filhos é denominado **nó interno** (exemplos: 1, 2, 6, 4, 7);
- Os nós que tem o mesmo pai, são **irmãos** (exemplos: 8 e 9; 2, 3, 4 e 5).



Estrutura de dados: Árvore



Tree data structure

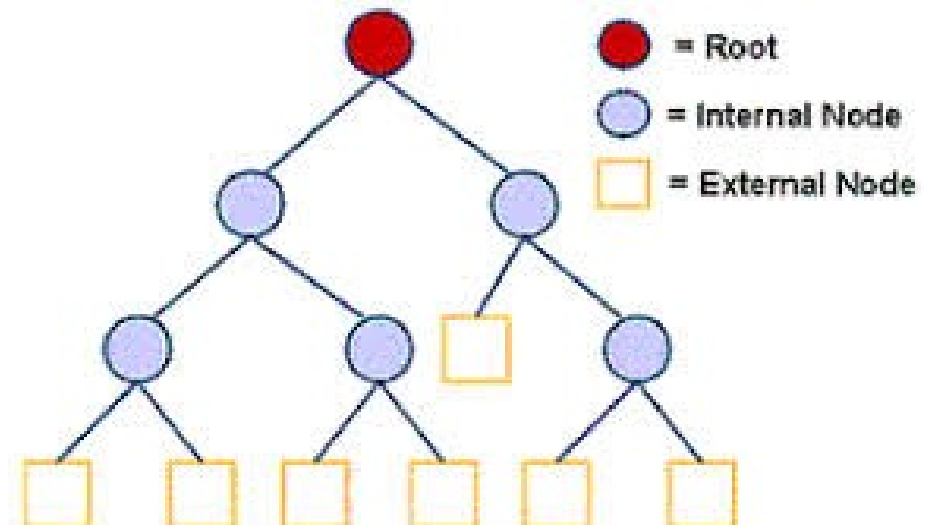


Level 0

Level 1

Level 2

Level 3

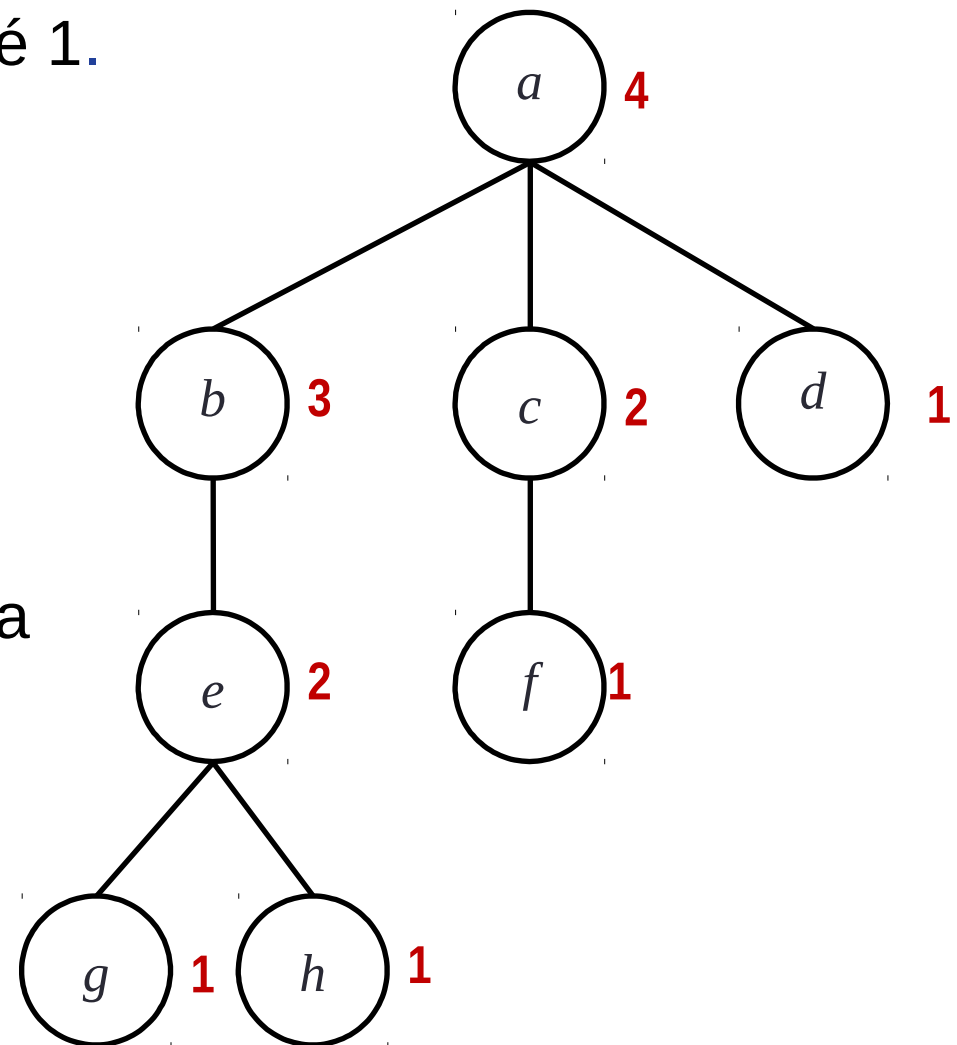


Altura de uma Árvore

- A **altura de um nó** é o número de nós que compõem os maiores caminhos (sempre descendentes) do nó a uma folha.
- Em particular, a altura duma folha é 1.

- A **altura duma árvore** vazia é 0;
- A **altura duma árvore** não vazia, é a altura do nó raiz.

Altura da árvore: 4

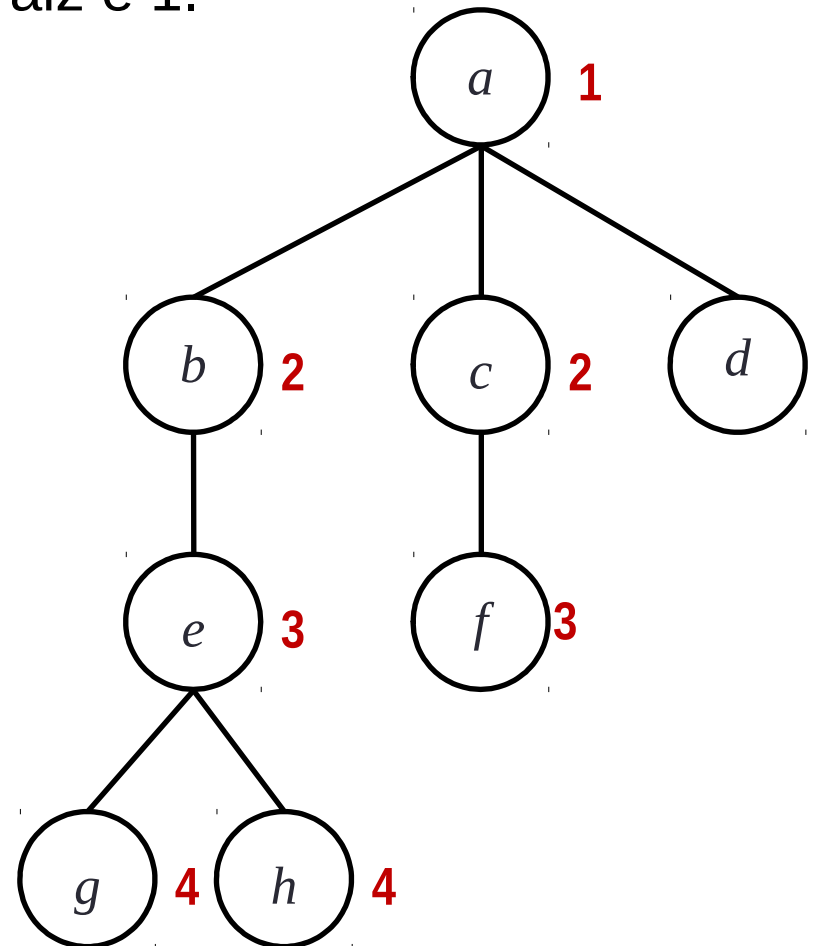


Profundidade de uma Árvore

- A **profundidade ou nível de um nó** é o número de nós do (único) caminho (sempre descendente) da raiz ao nó;
- Em particular, a profundidade ou nível da raiz é 1.

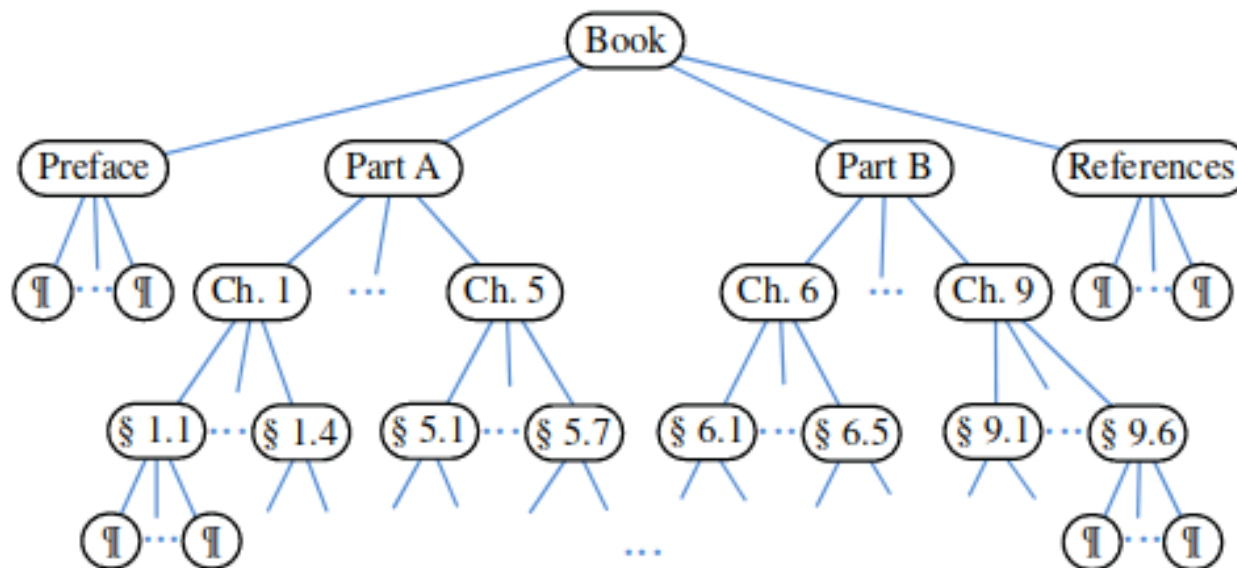
- A **profundidade duma árvore** vazia é 0;
- A **profundidade duma árvore** não vazia, é o máximo das profundidades das suas folhas.

Profundidade da árvore: 4



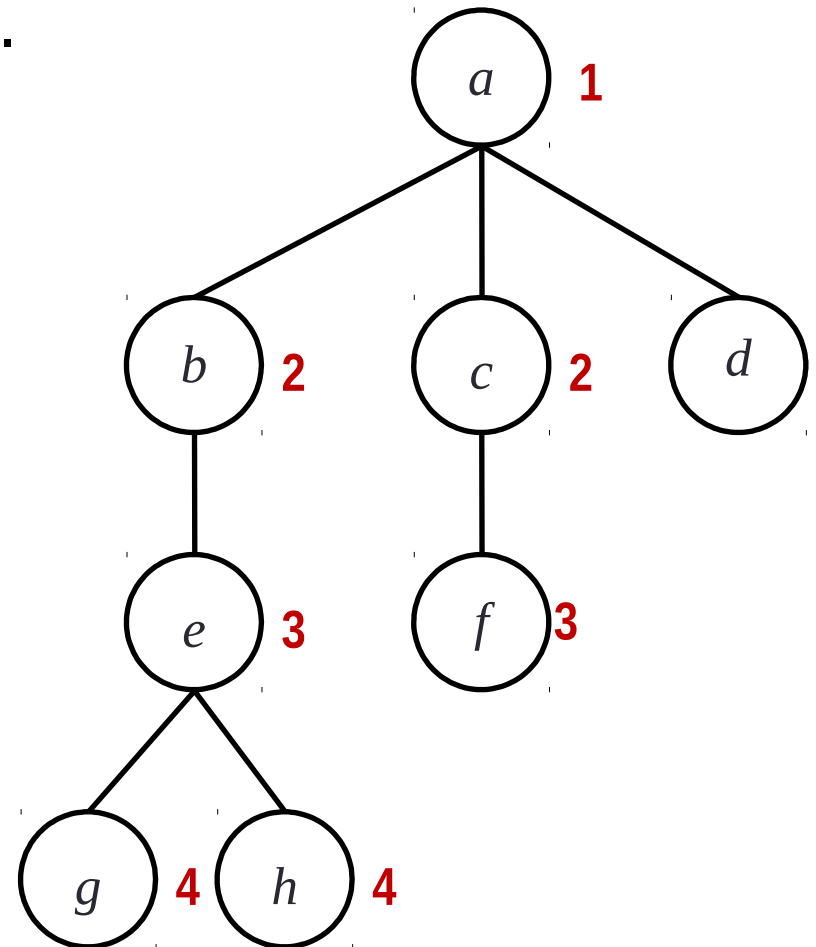
Árvores Ordenadas

- Uma árvore é ordenada se existe uma ordem linear para os filhos de cada nó: 1º filho, 2º filho, etc.
 - Normalmente, da esquerda para a direita.



Percurso numa árvore

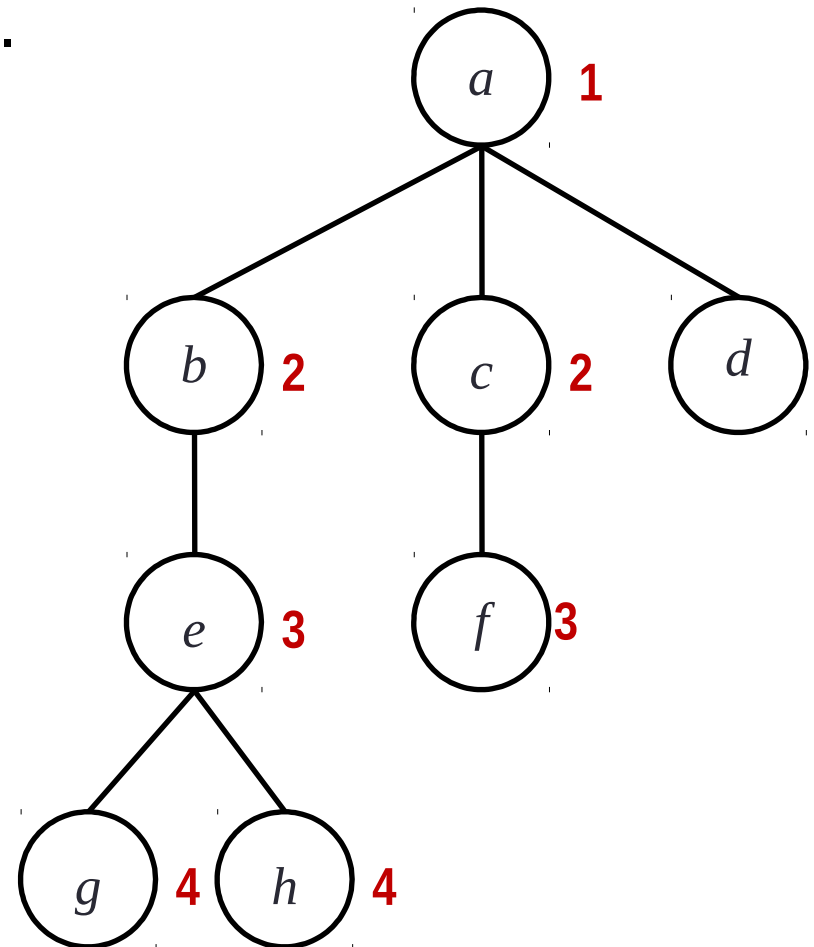
- Por níveis:
 - percurso denominado Breadth First Search (BFS), ou em português, busca em largura.



Percurso numa árvore

- Por níveis:
 - percurso denominado Breadth First Search (BFS), ou em português, busca em largura.

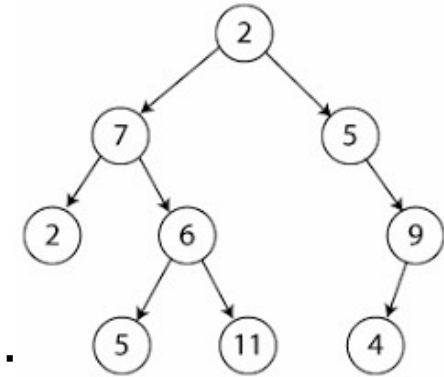
a; b; c; d; e; f; g; h



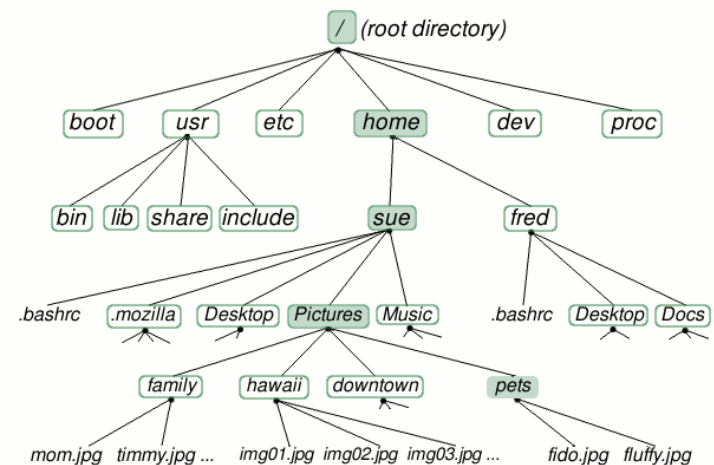
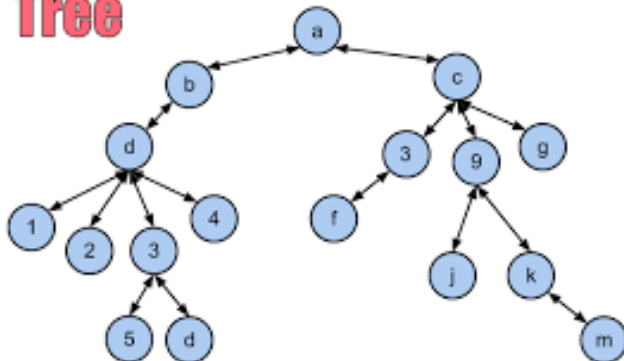
Classificação de Árvores

Classificação da Árvore quanto ao número de filhos:

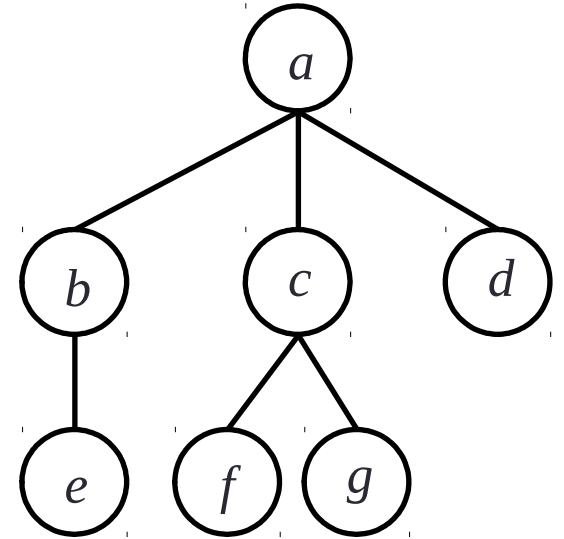
- **Árvore Binária:** todo o nó tem no máximo 2 filhos;
- **Árvore Ternária:** todo o nó tem no máximo 3 filhos;
- **Árvore N-ária:** todo o nó tem no máximo N filhos;
- **Árvore Genérica:** todo o nó tem um número (finito mas) ilimitado de filhos.



Tree

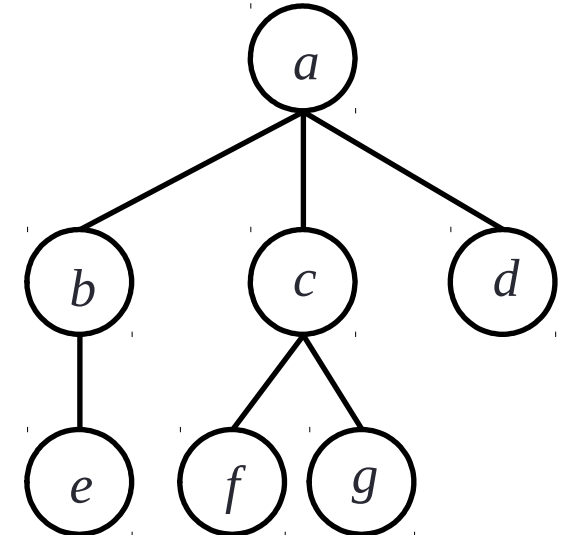
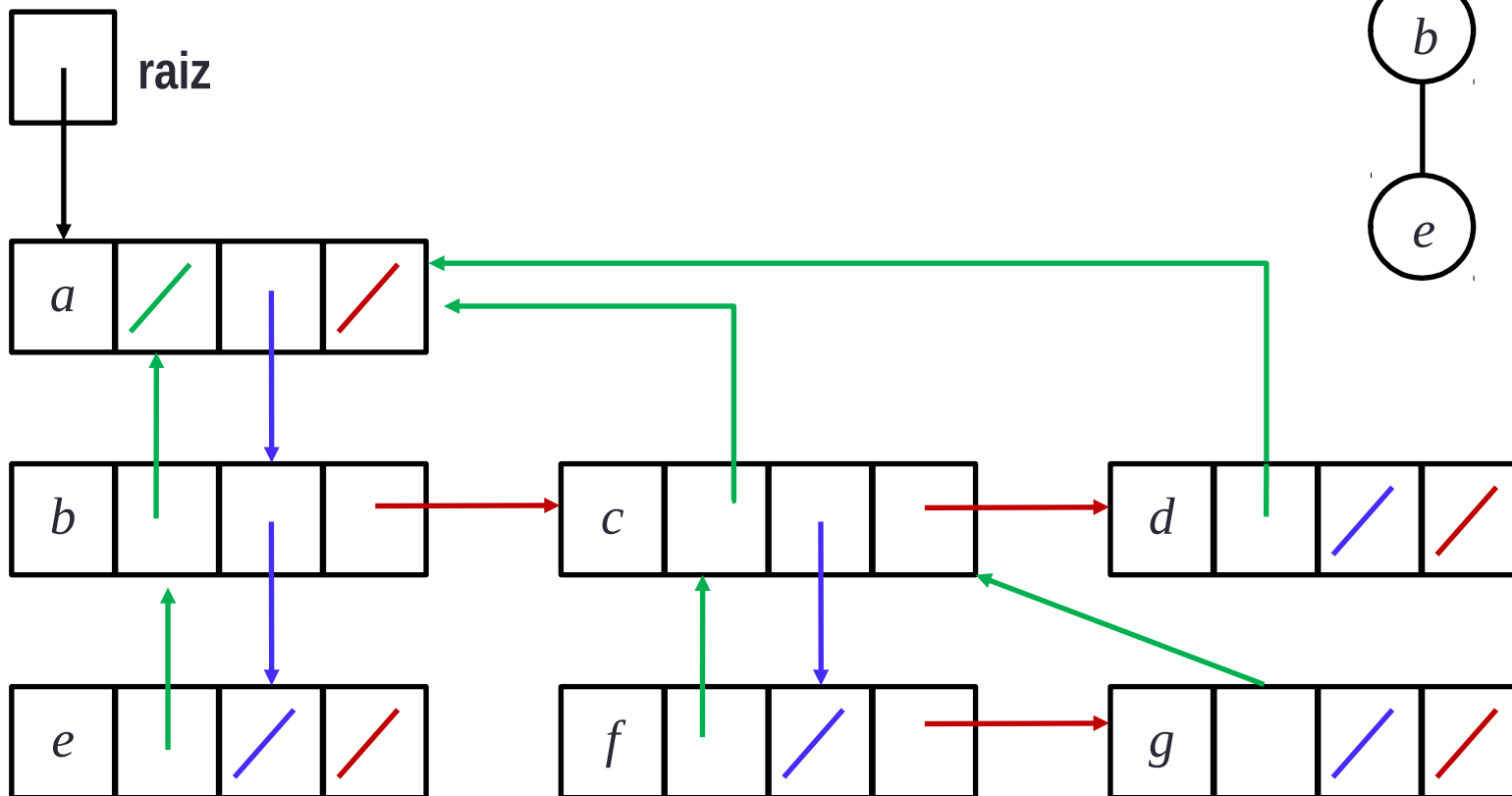


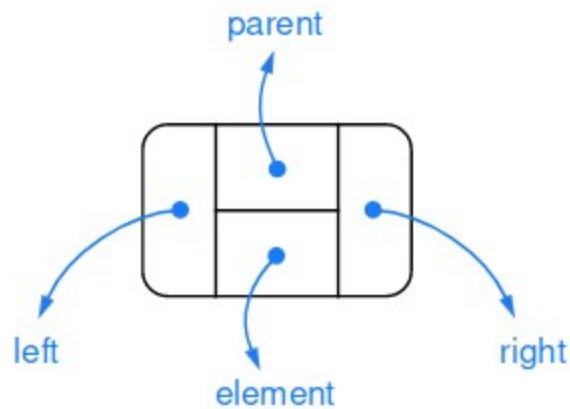
Como implementar Árvores Genéricas?



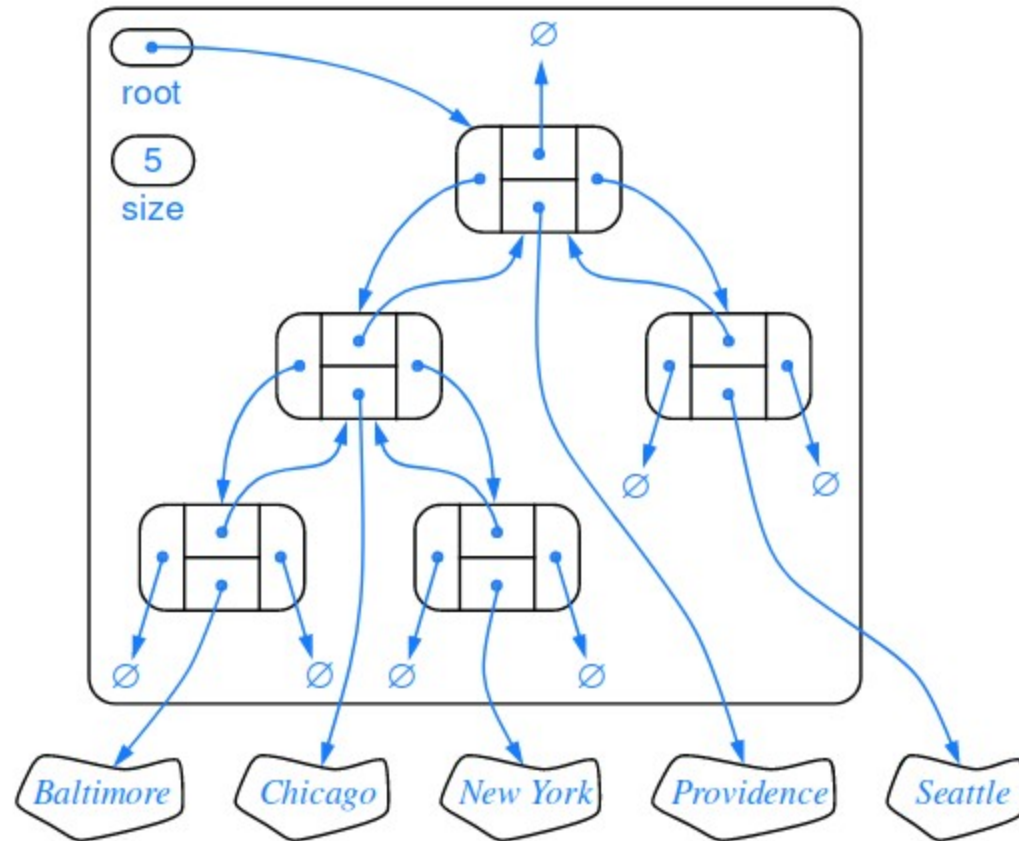
Como implementar Árvores Genéricas?

(elemento, **pai**, **filho esquerdo**, **irmão direito**)



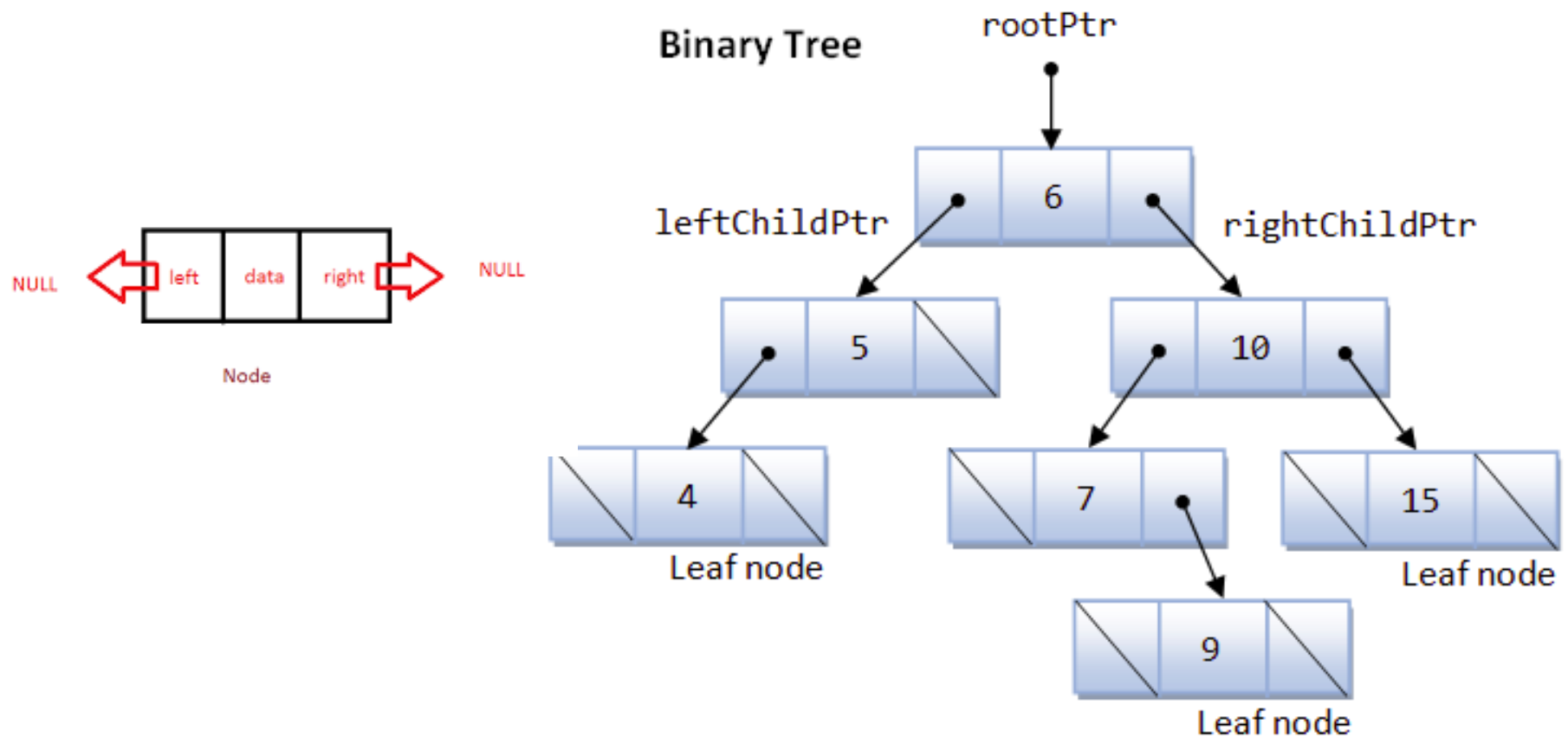



(a)



(b)

Como implementar Árvores Binárias? (2)





Interface Node<E>

Nó da árvore

- **Árvore** é um conjunto finito de **nós**, onde se encontram os elementos.

```
package dataStructures;
```

```
public interface Node<E> {  
    //returns the element stored at this node  
    E getElement();  
}
```

Classe abstracta Tree (1)

```
package dataStructures;

public abstract class Tree<E> {

    //The root
    protected Node<E> root;

    //Number of elements
    protected int currentSize;

    //Returns the node of the root of the tree (or null if empty).
    protected Node<E> root(){
        return root;
    }

    ...
}
```

Classe abstracta Tree (2)

```
package dataStructures;
```

```
public abstract class Tree<E> {
```

```
...
```

```
//Returns the parent of node n (or null if n is the root).
```

```
protected abstract Node<E> parent(Node<E> n);
```

```
//Returns an iterator containing the children of node n (if any).
```

```
protected abstract Iterator<Node<E>> children(Node<E> n);
```

```
//Returns the number of children of node n
```

```
protected abstract int numChildren(Node<E> n); //Number of elements
```

```
...
```

Classe abstracta Tree (3)

```
package dataStructures;
```

```
public abstract class Tree<E> {
```

```
    ...
```

```
    //Returns true if node n has at least one child.
```

```
    protected boolean isInternal(Node<E> n) {
```

```
        return numChildren(n)>=1;
```

```
    }
```

```
    //Returns true if node n does not have any children.
```

```
    protected boolean isExternal(Node<E> n) {
```

```
        return numChildren(n)==0;
```

```
    }
```

```
    //Returns true if node n is the root of the tree.
```

```
    protected boolean isRoot(Node<E> n) {
```

```
        return n==root;
```

```
    }
```

```
    ...
```

Classe abstracta Tree (4)

```
package dataStructures;
```

```
public abstract class Tree<E> {
```

```
...
```

```
//Returns the number of elements that are in the tree.
```

```
public int size() {  
    return currentSize;  
}
```

```
//Returns true if the tree does not contain any elements
```

```
public boolean isEmpty() {  
    return currentSize==0;  
}
```

```
//Returns an iterator for all elements in the tree
```

```
public abstract Iterator<E> iterator();
```

```
...
```

Classe abstracta Tree (5)

```
package dataStructures;
```

```
public abstract class Tree<E> {
```

```
    ...
```

```
    //Returns the height of the subtree rooted at Node n.
```

```
    protected int height(Node<E> n) {
```

```
        ...
```

```
    }
```

```
    //Returns the height of the tree.
```

```
    public int height() {
```

```
        if(isEmpty())
```

```
            return 0;
```

```
            return height(root);
```

```
    }
```

```
}
```

Classe abstracta Tree (5)

```
package dataStructures;
```

```
public abstract class Tree<E> {
```

```
    ...
```

```
    //Returns the height of the subtree rooted at Node n.
```

```
    protected int height(Node<E> n) {
```

```
        int h=1;
```

```
        if (isExternal(n))
```

```
            return h;
```

```
        Iterator <Node<E>> it =children(n);
```

```
        while (it.hasNext())
```

```
            h=Math.max(h,1+ height(it.next()));
```

```
        return h;
```

```
    }
```

```
    //Returns the height of the tree.
```

```
    public int height() {
```

```
        if(isEmpty())
```

```
            return 0;
```

```
        return height(root);
```

```
    }
```

```
}
```

Qual a Complexidade
Temporal



Classe abstracta Tree (6)

Qualquer filho é visitado. Quantos nós filhos temos?

$n-1$ nós,

já que todo o nó tem um pai (excepto a raiz)

$O(n)$

//Returns the height of the subtree rooted at Node n.

```
protected int height(Node<E> n) {  
    int h=1;  
    if (isExternal(n))  
        return h;  
    Iterator <Node<E>> it =children(n);  
    while (it.hasNext())  
        h=Math.max(h,1+ height(it.next()));  
    return h;  
}
```

//Returns the height of the tree rooted at Node n.

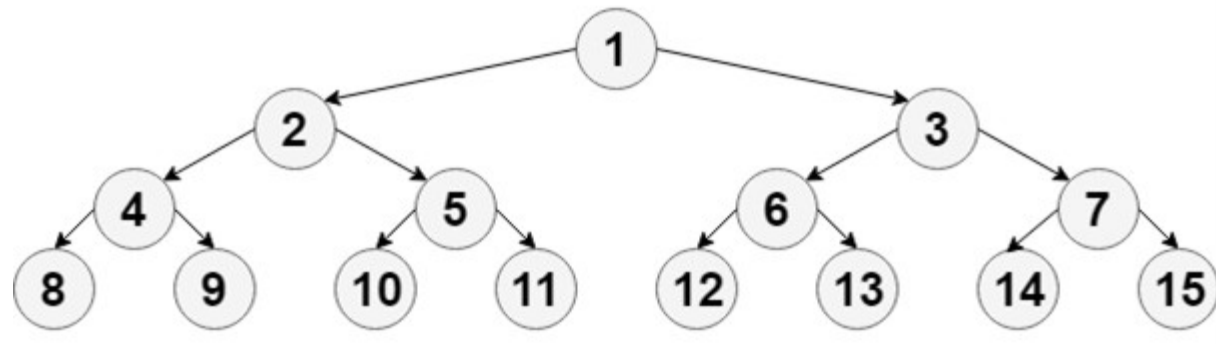
```
public int height() {  
    if(isEmpty())  
        return 0;  
    return height(root);  
}
```


Árvore Binária

Relação entre altura e número de nós (1)

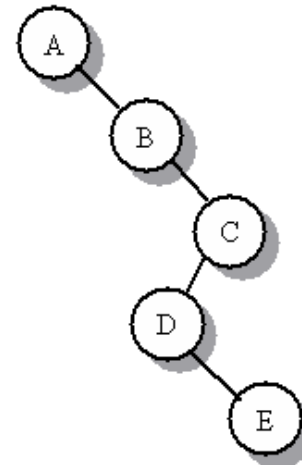
- Número de nós (elementos) numa árvore binária de altura h :
 - Se todo o nó (excepto as folhas) tem sempre 2 filhos, o número máximo de nós seria:

$$2^h - 1$$



- Se todo o nó (excepto as folhas) tem no máximo 1 filho, o número de nós é:

$$h$$

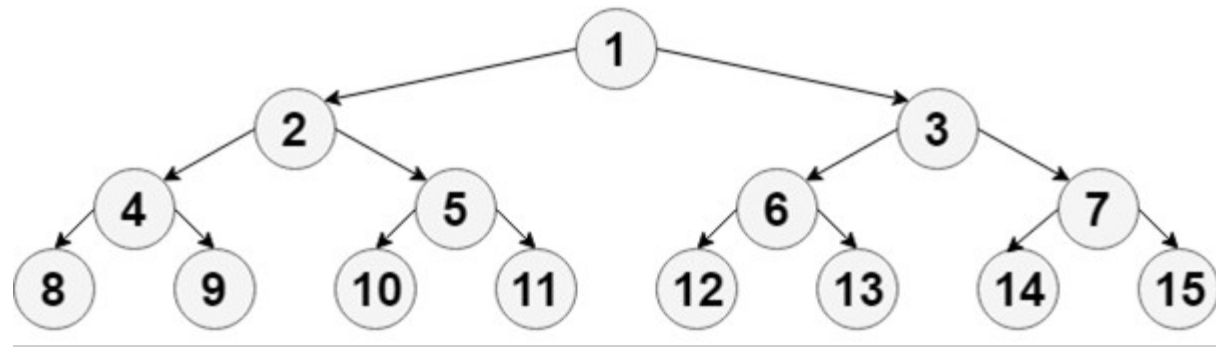


Árvore Binária

Relação entre altura e número de nós (2)

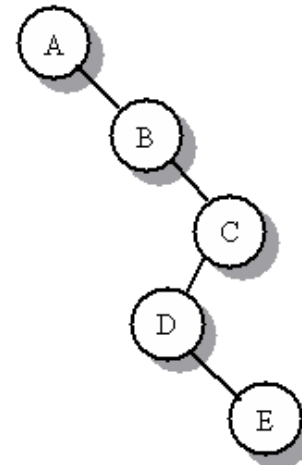
- Altura de uma árvore com n nós (elementos):
 - Se todo o nó (excepto as folhas) tem sempre 2 filhos, a altura da árvore:

$$\lceil \log(n+1) \rceil$$



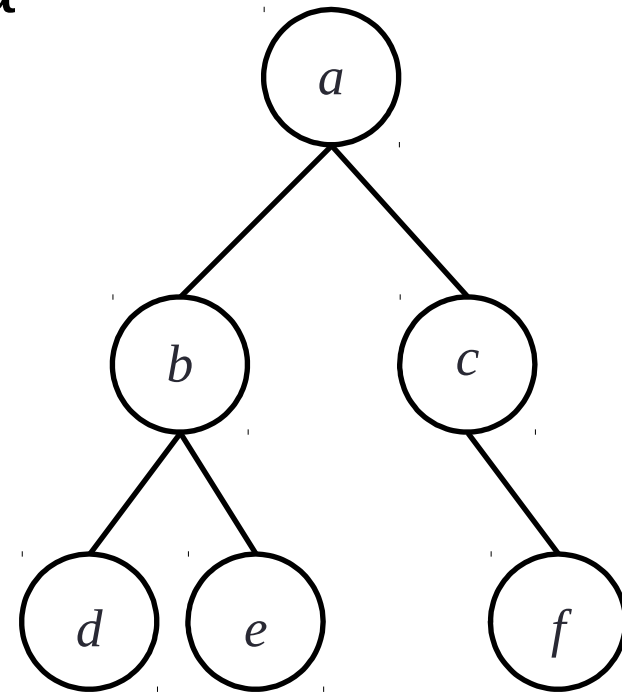
- Se todo o nó (excepto as folhas) tem no máximo 1 filho, a altura da árvore:

$$n$$



Percursos numa árvore ordenada binária (1)

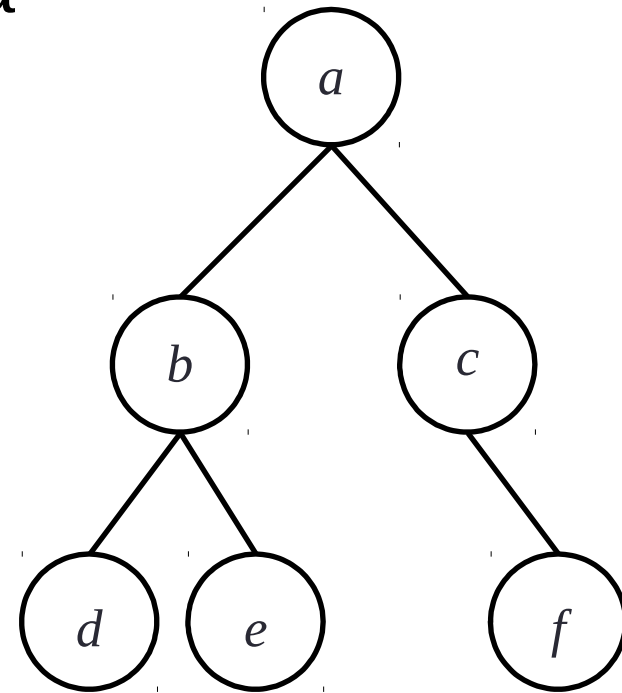
- Prefixo:
 - elemento da raiz; prefixo da sub-árvore esquerda; prefixo da sub-árvore direita



Percursos numa árvore ordenada binária (1)

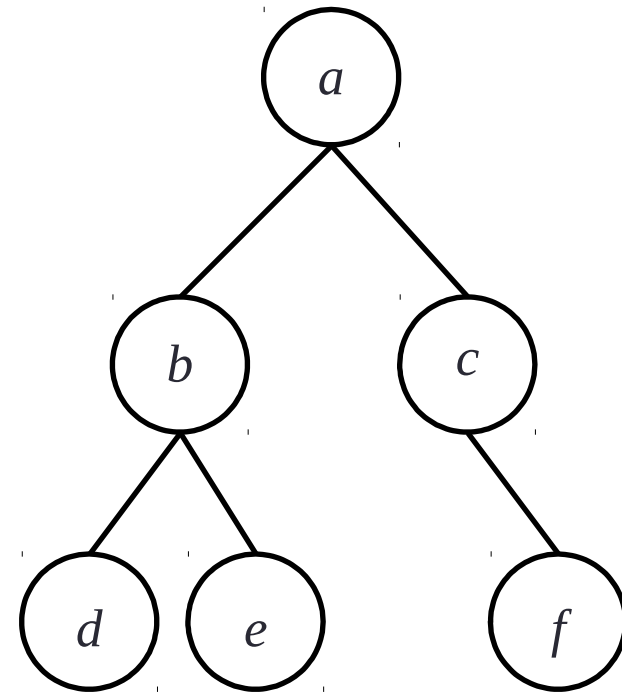
- Prefixo:
 - elemento da raiz; prefixo da sub-árvore esquerda; prefixo da sub-árvore direita

a; b; d; e; c; f



Percursos numa árvore ordenada binária (2)

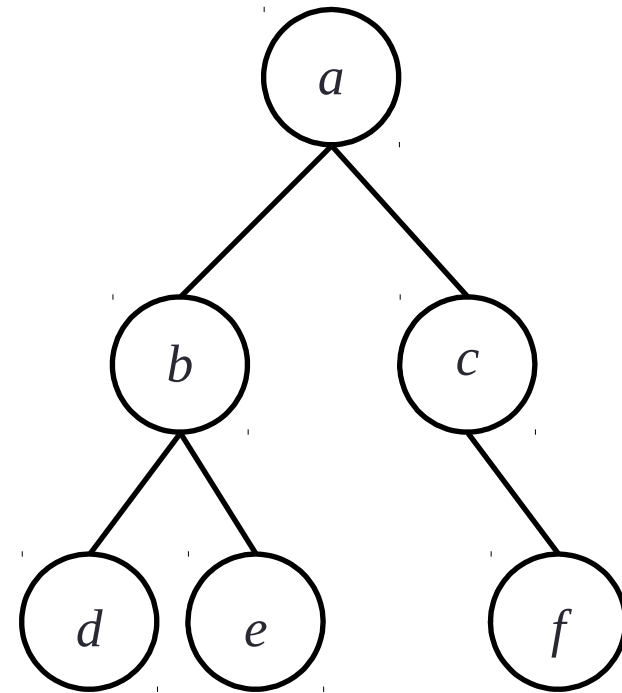
- **Infixo:**
 - infixo da sub-árvore esquerda; elemento da raiz; infixo da sub-árvore direita



Percursos numa árvore ordenada binária (2)

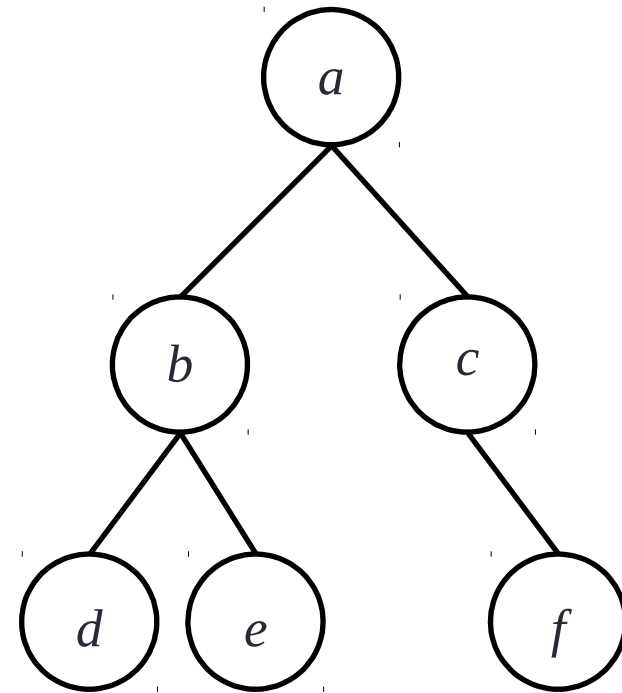
- **Infixo:**
 - infixo da sub-árvore esquerda; elemento da raiz; infixo da sub-árvore direita

d; b; e; a; c; f



Percursos numa árvore ordenada binária (3)

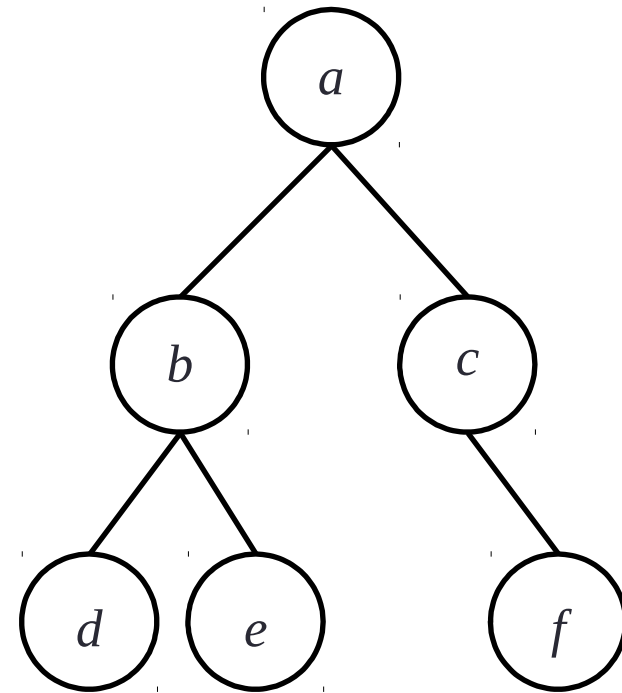
- Sufixo:
 - sufixo da sub-árvore esquerda; sufixo da sub-árvore direita; elemento da raiz



Percursos numa árvore ordenada binária (3)

- Sufixo:
 - sufixo da sub-árvore esquerda; sufixo da sub-árvore direita; elemento da raiz

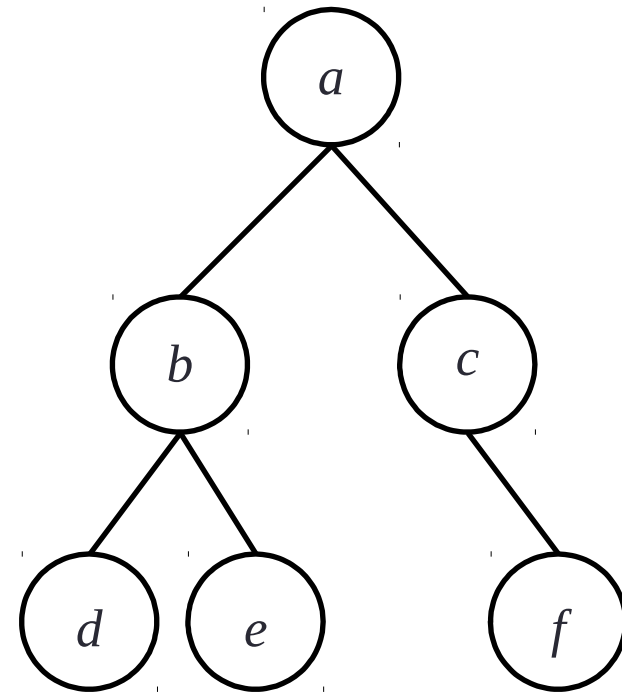
d; e; b; f; c; a



Percursos numa árvore ordenada binária (4)

- Por níveis:
 - percurso denominado breadth first search (BFS) ou em português, busca em largura.

a; b; c; d; e; f



Classe BinaryTree (1)

```
package dataStructures;
public abstract class BinaryTree<E> extends Tree<E> {

    //Returns the node of n's left child (or null if no child exists).
    protected abstract Node<E> left(Node<E> n);

    //Returns the node of n's right child (or null if no child exists).
    protected abstract Node<E> right(Node<E> n);

    //Returns the node of n's parent (or null if no parent exists).
    protected abstract Node<E> parent(Node<E> n);

    //Returns the node of n's sibling (or null if no sibling exists).
    protected Node<E> sibling(Node<E> n){
        ...
    }
    ...
}
```

Classe BinaryTree (1)

```
package dataStructures;
public abstract class BinaryTree<E> extends Tree<E> {

    //Returns the node of n's left child (or null if no child exists).
    protected abstract Node<E> left(Node<E> n);

    //Returns the node of n's right child (or null if no child exists).
    protected abstract Node<E> right(Node<E> n);

    //Returns the node of n's parent (or null if no parent exists).
    protected abstract Node<E> parent(Node<E> n);

    //Returns the node of n's sibling (or null if no sibling exists).
    protected Node<E> sibling(Node<E> n){
        Node<E> parent=parent(n);
        if (parent==null) //the root
            return null;
        if (n==left(parent))
            return right(parent);
        return left(parent);
    }
    ...
}
```

Classe BinaryTree (2)

```
package dataStructures;
```

```
public abstract class BinaryTree<E> extends Tree<E> {
```

```
    ...
```

```
    //Returns the number of children of node n
```

```
    protected int numChildren(Node<E> n) {
```

```
        ...
```

```
    }
```

```
    //Returns an iterator containing the children of node n (if any).
```

```
    protected Iterator<Node<E>> children(Node<E> n){
```

```
        ...
```

```
    }
```

```
}
```

Classe BinaryTree (2)

```
package dataStructures;
```

```
public abstract class BinaryTree<E> extends Tree<E> {
```

```
    ...
```

```
    //Returns the number of children of node n
```

```
    protected int numChildren(Node<E> n) {
```

```
        int count=0;
```

```
        if (left(n)!=null) count++;
```

```
        if (right(n)!=null) count++;
```

```
        return count;
```

```
    }
```

```
    //Returns an iterator containing the children of node n (if any).
```

```
    protected Iterator<Node<E>> children(Node<E> n){
```

```
        ...
```

```
    }
```

```
}
```

Classe BinaryTree (2)

```
package dataStructures;
```

```
public abstract class BinaryTree<E> extends Tree<E> {
```

```
    ...
```

```
    //Returns the number of children of node n
```

```
    protected int numChildren(Node<E> n) {
```

```
        int count=0;
```

```
        if (left(n)!=null) count++;
```

```
        if (right(n)!=null) count++;
```

```
        return count;
```

```
    }
```

```
    //Returns an iterator containing the children of node n (if any).
```

```
    protected Iterator<Node<E>> children(Node<E> n){
```

```
        if (isExternal(n))
```

```
            return null;
```

```
        List<Node<E>> aux = new SinglyLinkedList<Node<E>>(2);
```

```
        if (left(n)!=null)
```

```
            aux.addLast(left(n));
```

```
        if (right(n)!=null)
```

```
            aux.addLast(right(n));
```

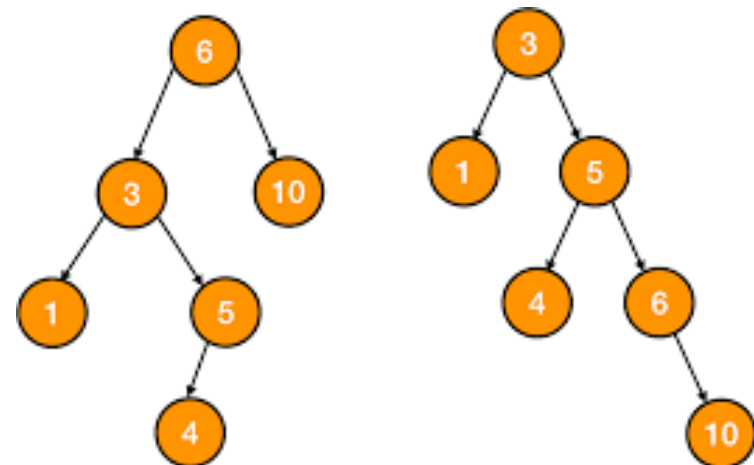
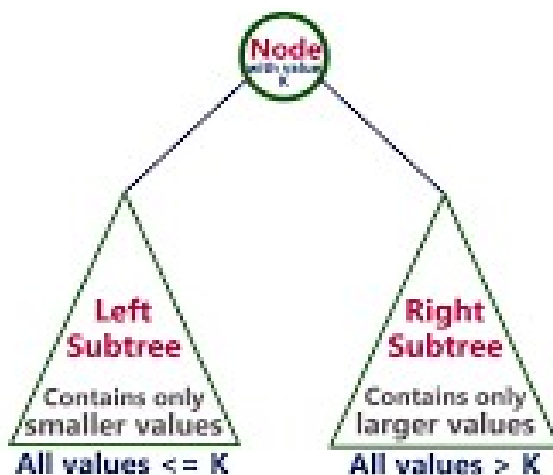
```
        return aux.iterator();
```

```
    }
```

```
}
```

Árvore binária de pesquisa

- Uma árvore binária de pesquisa é uma árvore ordenada binária em que todo nó X verifica as seguintes propriedades:
 - - Todo nó na **sub-árvore esquerda**, contem um elemento **menor** que o contido em X;
 - - Todo nó na **sub-árvore direita**, contem um elemento **maior** que o contido em X.



Two Binary Search Trees Representing Same Set