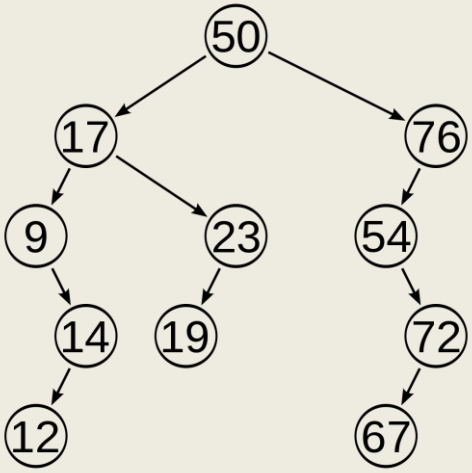


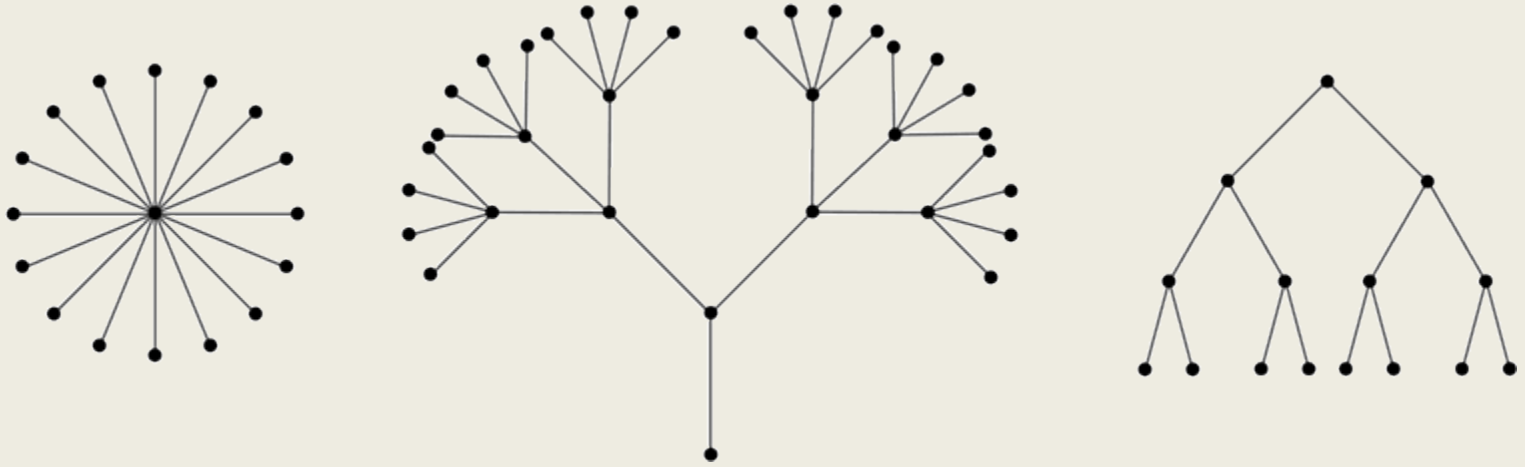
Árvores

- Terminologia
- Travessias
- Árvores Binárias
- Estruturas de dados para representar árvores

Árvores (Matemática)

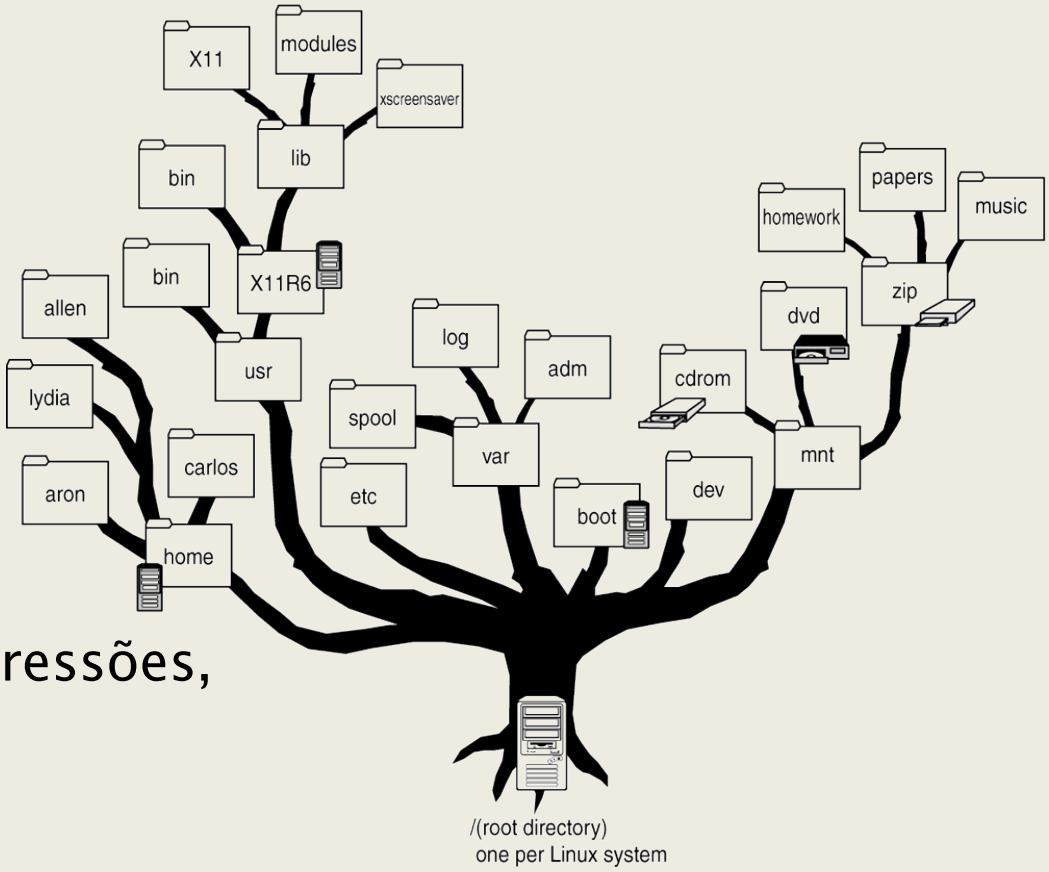
- Um conceito matemático parte da teoria de grafos

- Neste contexto, chama-se **árvore** a um grafo que tem quaisquer dois nós ligados por um único caminho



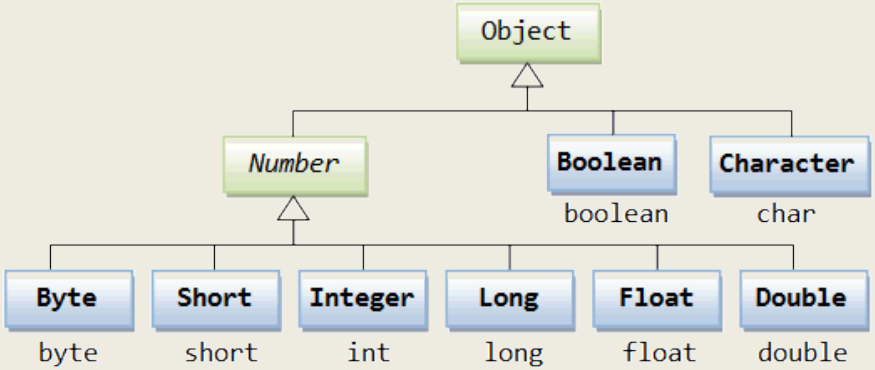
Árvores (Computação)

- Na computação, as árvores têm adicionalmente uma raiz
- Permitem representar elementos de uma forma hierárquica
  - sistema de ficheiros
  - organogramas
  - tipos Java

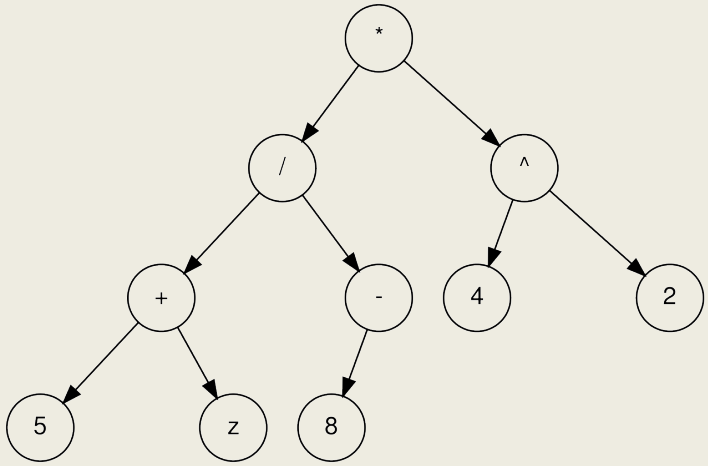


- Permitem representar expressões, até programas

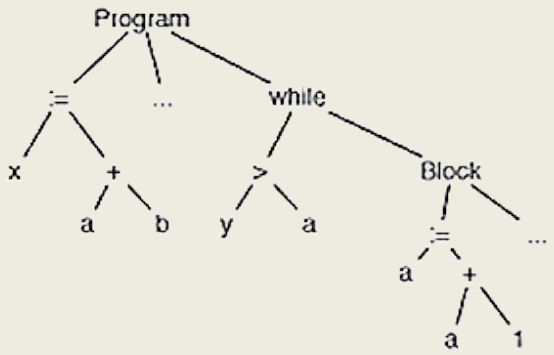
Exemplos



$[(5+z)/(-8)] * 4^2$




```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```



Árvores

- Características:
  - uma árvore tem um conjunto de **nós**
  - uma árvore com um conjunto de nós não vazio tem um nó distinguido a que se chama **raiz**
  - com excepção da raiz, cada nó na árvore tem um único **pai**
  - cada nó tem zero ou mais **filhos**
- Se adicionalmente a ordem dos filhos em cada nó é relevante então temos uma **árvore ordenada** (***ordered tree***)
- Uma **árvore binária** (***binary tree***) é uma árvore ordenada em que cada nó tem no máximo dois filhos, sendo que cada um desses filhos é identificado como sendo o filho esquerdo ou direito.

Terminologia e Propriedades

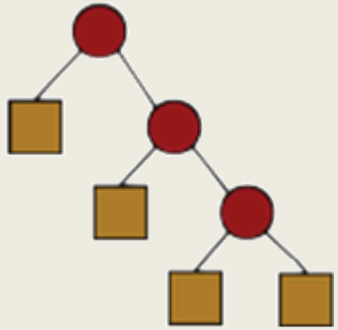
- **Terminologia:**
  - **Nó externo ou folha**: nó sem filhos
  - **Nó interno**: tem pelo menos um filho
  - **Nós irmãos**: têm o mesmo pai
  - **Antepassados**: o próprio, o pai, o pai do pai, ...
  - **Descendentes:** o próprio, os filhos, os filhos dos filhos,...
  - **Profundidade de um nó:** número de antepassados
  - **Altura da árvore:** profundidade máxima dos nós
  - **Aresta:** um par de nós (u,v) em que u é pai de v
  - **Caminho:** sequência de nós ligados por arestas
  - **Sub-árvore:** uma árvore consistindo num nó e todos os seus descendentes
- Seja *n* o número de nós e *h* a altura. Numa árvore binária temos:

$$h \leq n \leq 2^h - 1$$

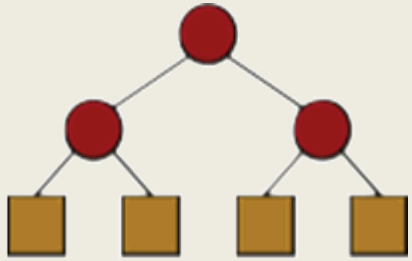
Árvores Binárias Cheias

Seja ***n*** o número de nós, ***f*** o número de folhas e ***h*** a altura

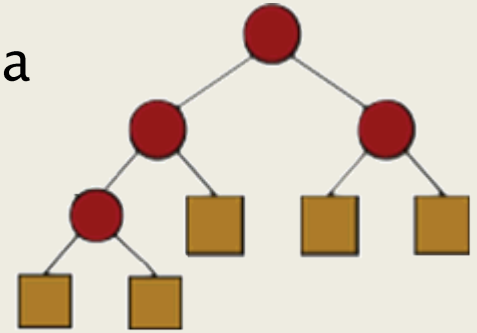
- Uma árvore binária é **cheia** se cada nó tem 0 ou 2 filhos
  - $n = 2*f-1$



- Uma árvore cheia é **perfeita** se todas as folhas têm a mesma profundidade
  - $n = 2^h-1$
  - $f = 2^{h-1}$



- Uma árvore binária é **completa** se é perfeita até ao nível *h-1* e o nível *h* está preenchido a partir da esquerda

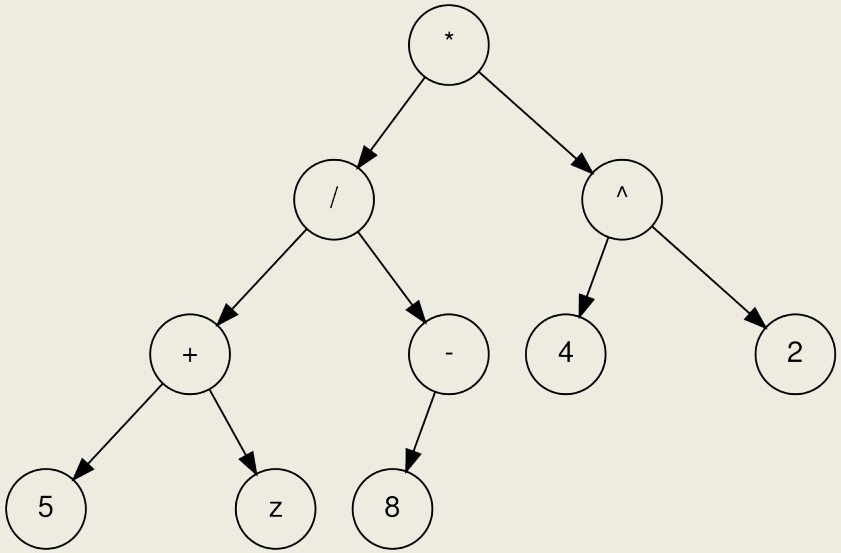


Travessia

- Muitos algoritmos sobre árvores recorrem à travessia dos nós de uma forma sistemática, seguindo uma determinada ordem

- Exemplo:

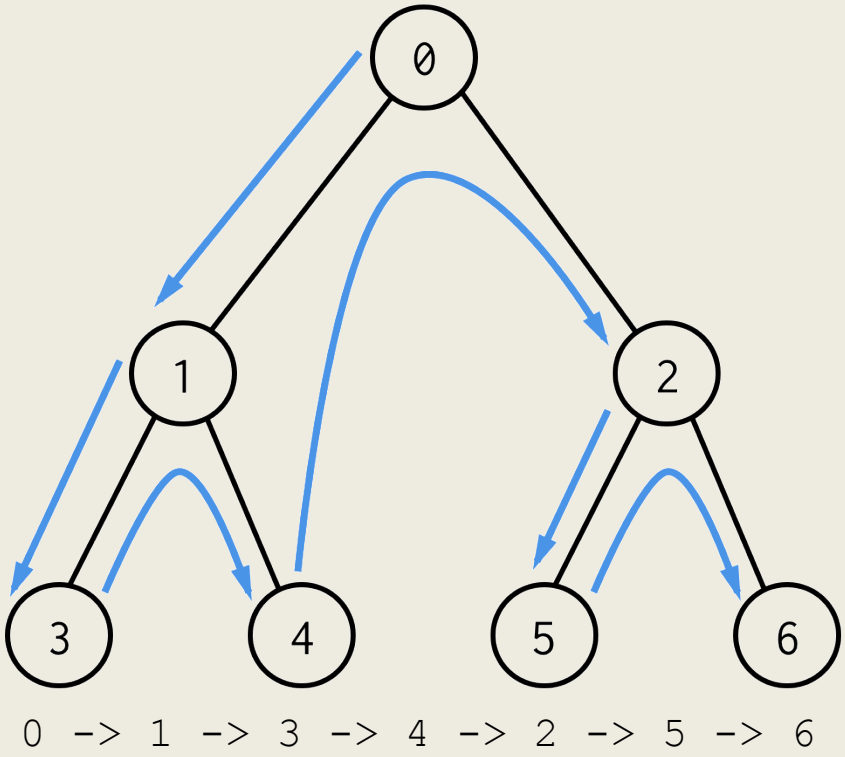
$[(5+z)/(-8)] * 4^2$



Travessia

- Entre as travessias em profundidade (***depth-first***) temos:
  - **Travessia prefixa** (***preorder traversal***): cada nó é visitado antes dos seus descendentes

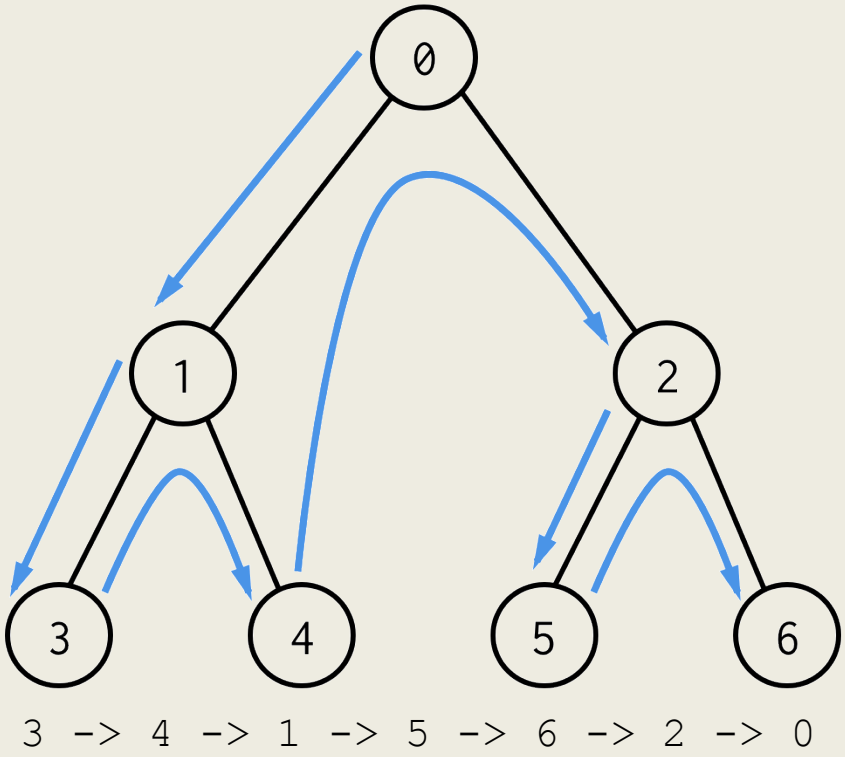
```
Algorithm preOrder(v) {
   visit(v)
   for each child w of v
      preOrder(w)
}
```



Travessia

- **Travessia sufixa** (***postorder traversal***): cada nó é visitado depois dos seus descendentes

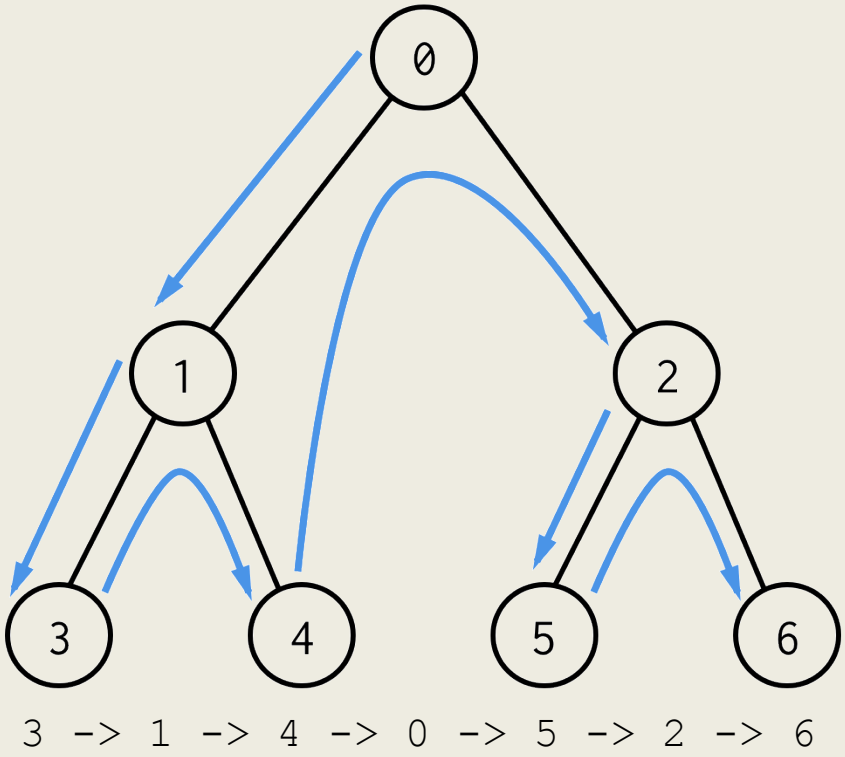
```
Algorithm postOrder(v) {
   for each child w of v
      postOrder(w)
   visit(v)
}
```



Travessia

- No caso de árvores binárias temos ainda outra travessia:
  - **Travessia infixa** (***inorder traversal***): visitada a sub-árvore esquerda, a raiz, depois a sub-árvore direita

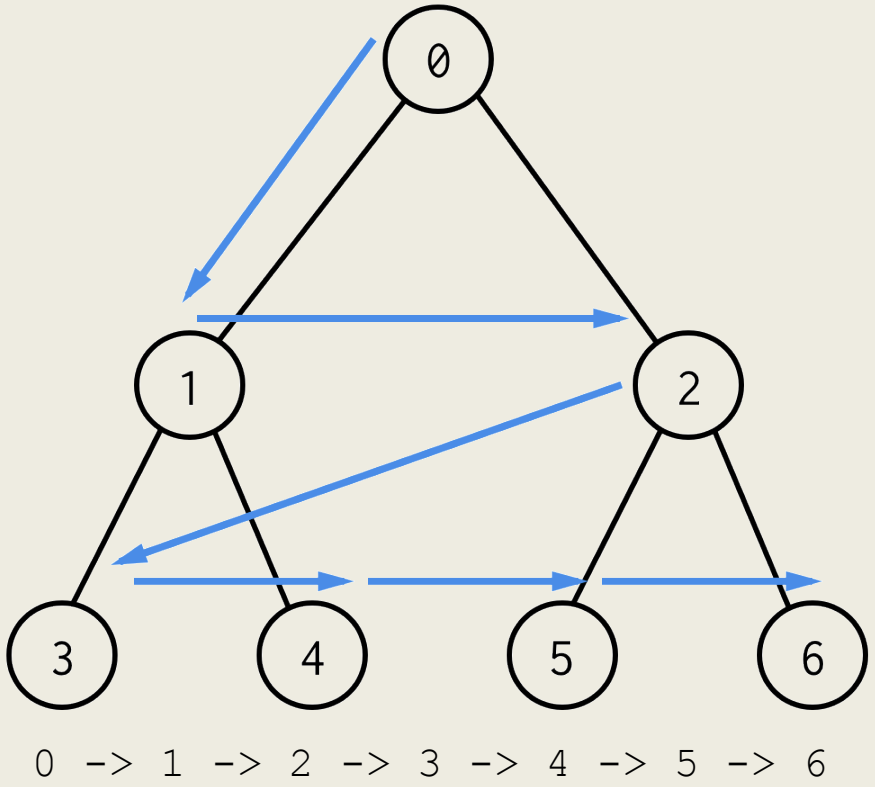
```
Algorithm inOrder(v) {
  if v has a left child l
    inOrder(l)
  visit(v)
  if v has a right child r
    inOrder(r)
}
```



Travessia

- Existe ainda a travessia em largura (***breadh-first***):

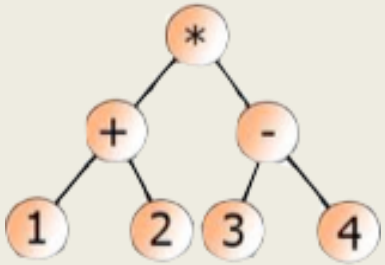
```
Algorithm breadth(tree) {
  q = empty queue
  enqueue root(tree) in q
  while (q is not empty)
    v = dequeue q
    visit(v)
    for each child w of v
      enqueue w in q
}
```

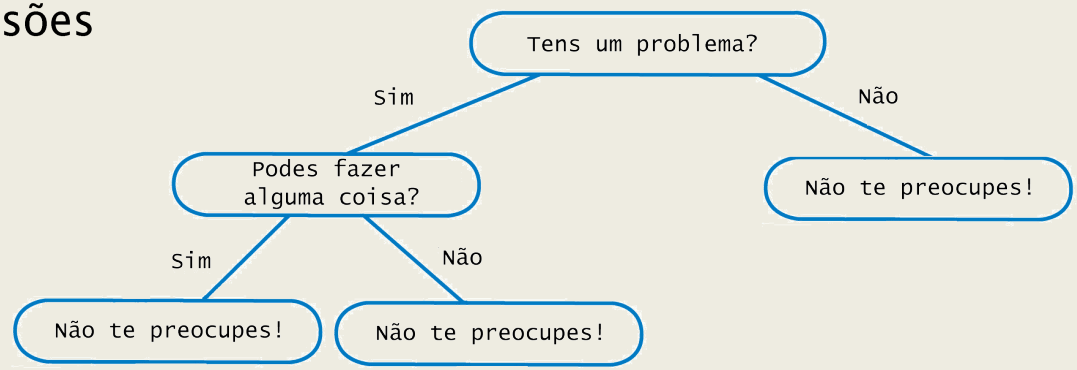


Árvores Binárias

- A partir de agora vamos concentrar-nos no ADT **árvore binária** (***binary tree***) e suas implementações

- Exemplos de aplicação:
  - Expressões aritméticas
    - nós internos são operadores
    - folhas são operandos
  - Processos de decisão
    - nós internos são perguntas de resposta sim/não
    - folhas são decisões





Árvore Binária

Este ADT pode ser definido recursivamente:

- uma árvore binária é vazia ou
- consiste num nó — a raiz — uma sub-árvore direita e uma sub-árvore esquerda

com as operações

- **empty**: constrói BT vazia
- **make(e,l,r)**: constrói uma árvore tendo
  - na raiz um nó com o elemento **e** e
  - **l**, **r** como sub-árvores esquerda e direita, respectivamente
- **leftSubtree**: indica a sub-árvore esquerda
- **rightSubtree**: indica a sub-árvore direita
- **data**: indica o elemento que está na raiz
- **isEmpty**: indica se é a BT vazia

Binary Tree: Especificação

```
specification BinaryTree[Element]
  sorts
    BinaryTree[Element]
  constructors
    empty: --> BinaryTree[Element];
    make: Element BinaryTree[Element] BinaryTree[Element] --> BinaryTree[Element];
  observers
    isEmpty: BinaryTree[Element];
    data: BinaryTree[Element] -->? Element;
    leftSubtree: BinaryTree[Element] -->? BinaryTree[Element];
    rightSubtree: BinaryTree[Element] -->? BinaryTree[Element];
  domains
    T: BinaryTree[Element];
    data(T)         if not isEmpty(T);
    leftSubtree(T)  if not isEmpty(T);
    rightSubtree(T) if not isEmpty(T);
  axioms
    T, T1, T2: BinaryTree[Element]; E: Element;
    data(make(E, T1, T2)) = E;
    leftSubtree(make(E, T1, T2)) = T1;
    rightSubtree(make(E, T1, T2)) = T2;
    isEmpty(empty());
    not isEmpty(make(E, T1, T2));
end specification
```

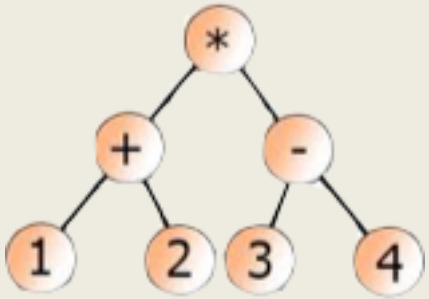
Binary Tree: Especificação

```
        ...
others
  isLeaf: BinaryTree[Element];
  height: BinaryTree[Element] --> int;
  occurrences: BinaryTree[Element] Element --> int;
  isBalanced: BinaryTree[Element];
  isPerfect: BinaryTree[Element];
  isComplete: BinaryTree[Element];
axioms
  T, T1, T2: BinaryTree[Element]; E: Element;
  isLeaf(T) iff not isEmpty(T) and isEmpty (leftSubtree(T)) and
         isEmpty(rightSubtree (T));
  height(empty ()) = 0;
  height(make(X, T1, T2)) = 1 + max (height (T1), height(T2));
  isBalanced(empty());
  isBalanced(make(E, T1, T2)) if isBalanced(T1) and isBalanced(T2) and
               abs(height(T1) - height(T2)) <= 1;
  isPerfect(empty());
  isPerfect(make(E, T1, T2)) if height(T1) = height(T2) and isPerfect(T1) and
            isPerfect(T2);
   isComplete (empty());
   isComplete (make(E, T1, T2)) if height(T1) = height(T2) and isPerfect(T1) and
            isComplete(T2) or height(T1) = height(T2) + 1 and isComplete(T1) and
            isPerfect(T2);
```

Exemplo

Construção de uma Árvore de Expressões

Algoritmo que, recorrendo a uma **pilha**, constrói uma árvore que representa uma expressão



((1+2)*(3-4))

Uma expressão é vista como uma sequência de *tokens* que se dividem em três tipos: números, operadores e parêntesis curvos.

Hipótese: a sequência representa efectivamente uma expressão.

Exemplo

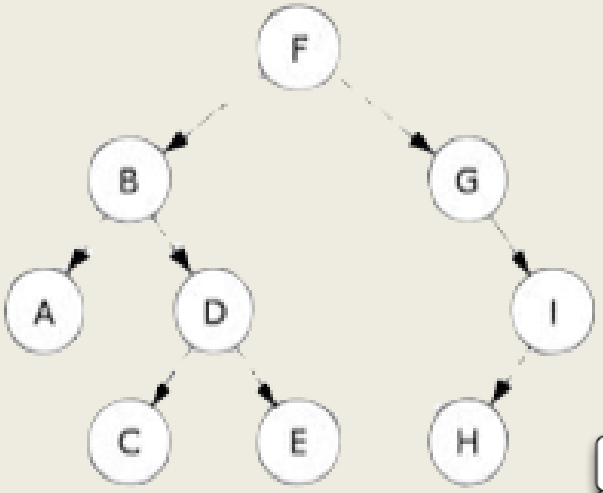
```
Algorithm buildExpression(X,n){
  S = make()   // stack of binary trees
  for i=0 to n-1 do
     if X[i] is a number or operator then
        T = make(X[i],empty(),empty())
        S = push(S,T)
     else if X[i] is ")" then
        T2 = peek(S); S = pop(S)
        T  = peek(S); S = pop(S)
        T1 = peek(S); S = pop(S)
        T  = make(data(T),T1,T2)
        S  = push(S,T)
  return peek(S)
}
```

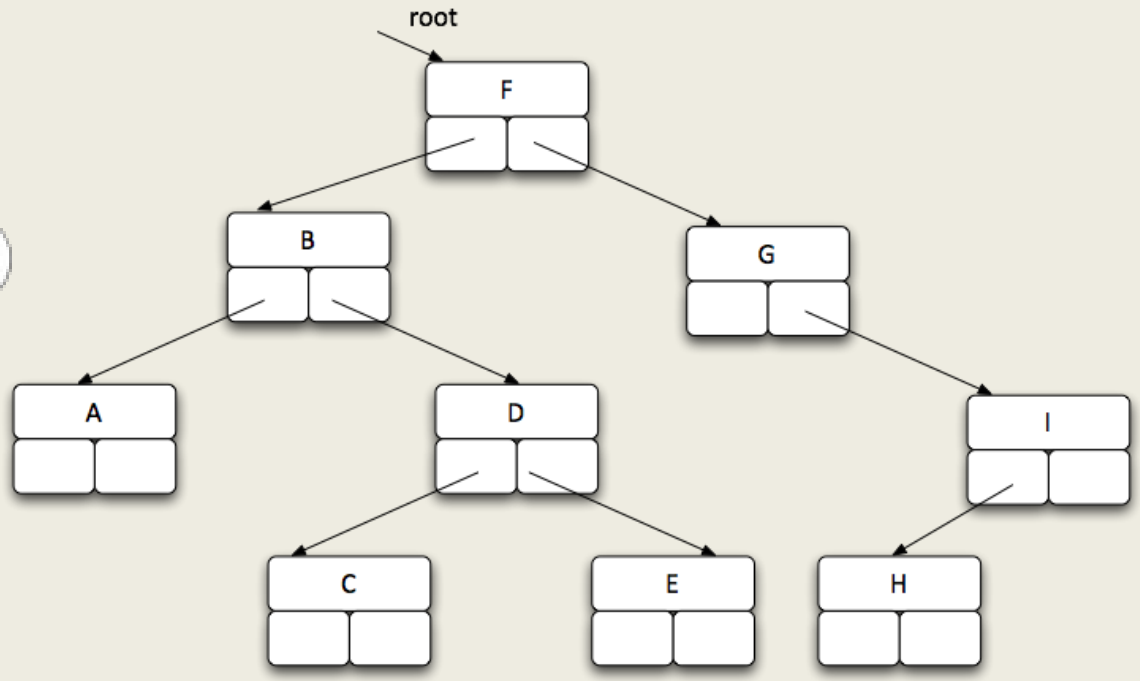
Binary Tree: Interface para Implementações Imutáveis

```
interface BinaryTree<E> {
   /** @return the data in the root node
    *  @requires !isEmpty()
    */
   public E data();
   /** @return the left subtree
    *  @requires !isEmpty()
    */
   public BinaryTree<E> leftSubtree();
   /** @return the right subtree
    *  @requires !isEmpty()
    */
   public BinaryTree<E> rightSubtree();
   /** @return If the tree is empty */
   public boolean isEmpty();
   /** @return The height of the tree */
   public int height();
   /** @param e an element
    *  @return The number of times e occurs in this tree
    */
   public int occurrences(E e);
}
```

Binary Tree: Implementação com estrutura ligada

- Escolhendo uma **estrutura ligada** a ideia é:
  - ter nós ligados às sub-árvores direita e esquerda
  - manter em *root* uma referência para o nó raiz da estrutura



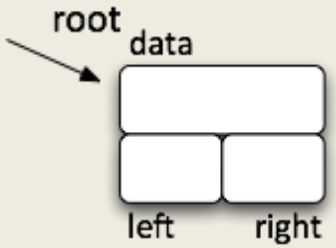


Binary Tree: Implementação com estrutura ligada

- `LinkedBinaryTree`: implementação de árvores binárias **imutáveis** com uma estrutura ligada

```
//class Node
private static class Node<E> {
   E data;
   Node<E> left;
   Node<E> right;
   private Node(E data, Node<E> left, Node<E> right ){
      this.data  = data;
      this.left  = left;
      this.right = right;
}
```

```
//the root node
private  Node<E> root;
```



Binary Tree: Implementação com estrutura ligada

```
// Constructors
public LinkedBinaryTree () {
  this.root = null;
}
//@requires left!=null && right!=null
public LinkedBinaryTree (E data, LinkedBinaryTree<E> left,
                                 LinkedBinaryTree<E> right) {
  this.root = new Node<E>(data, left.root, right.root);
}
//additional constructor
public LinkedBinaryTree (E data) {
  this.root = new Node<E>(data, null, null);
}
//auxiliary private construtor
private LinkedBinaryTree (Node<E> root) {
  this.root = root;
}
```

Binary Tree: Implementação com estrutura ligada

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

//@ requires !isEmpty()
public E data() {
  return root.data;
}

//@ requires !isEmpty()
public LinkedBinaryTree<E> leftSubtree() {
  return new LinkedBinaryTree<E> (root.left);
}

...
```

Binary Tree: Implementação com estrutura ligada

```
/**
 * The height of this tree.
 */
public int height() {
  return height (root);
}

private int height(Node<E> node) {
  return node == null? 0 :
          1 + Math.max(height(node.left), height(node.right));
}
```

Binary Tree: Implementação com estrutura ligada

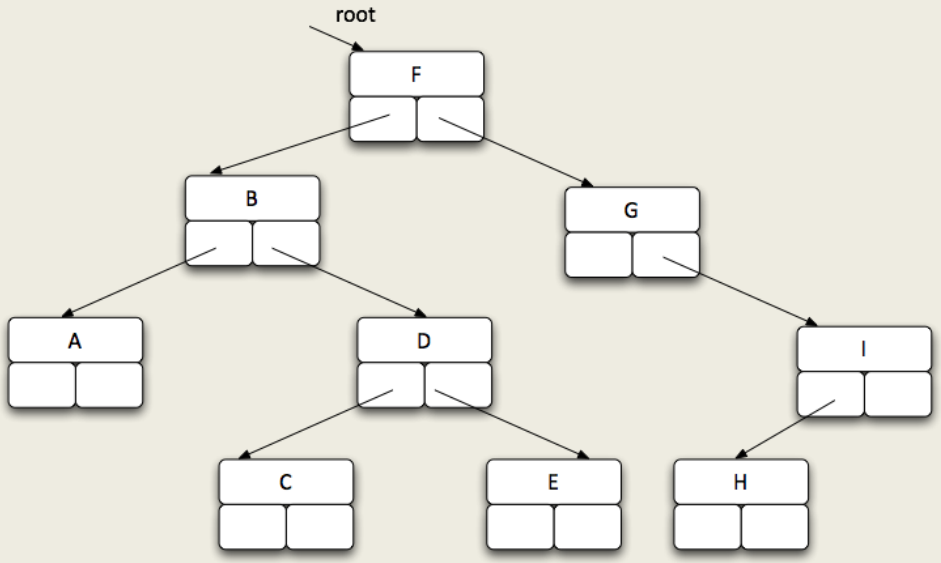
```
// The number of times an element occurs in this tree
public int occurrences(E element) {
  return occurrences(element, root);
}

private int occurrences(E element, Node<E> node) {
  if (node == null)
    return 0;
  else if (Objects.equals(element, node.data))
    return 1 + occurrences(element, node.left) +
               occurrences(element, node.right);
  else
    return occurrences(element, node.left) +
           occurrences(element, node.right);
}
```

Binary Tree: Implementação com estrutura ligada

Espaço ocupado é O(n)

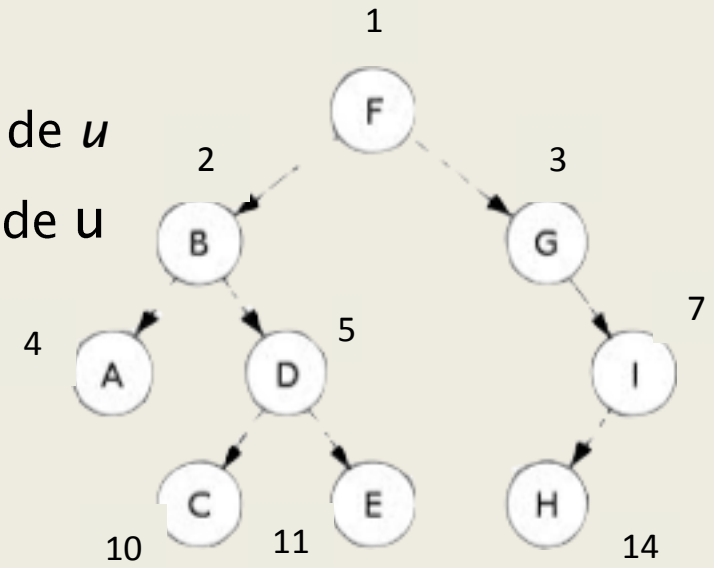
Method	Time
data	O(1)
left/rightSubtree	O(1)
isEmpty	O(1)
height	O(n)*
occurrences	O(n)

* O(1) se for usado atributo



Binary Tree: Implementação baseada num vector

- Escolhendo um **vector** a ideia é baseada numa forma de numerar os nós da árvore.
- Para todo o nó *v*, seja *p(v)* tal que
  - *p(v)=1* se *v* é a raiz
  - *p(v)=2*p(u)* se *v* é o filho esquerdo de *u*
  - *p(v)=2*p(u)+1* se *v* é o filho direito de u



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	F	B	G	A	D		I			C	E			H	

Espaço ocupado é no pior caso $O(2^n)$