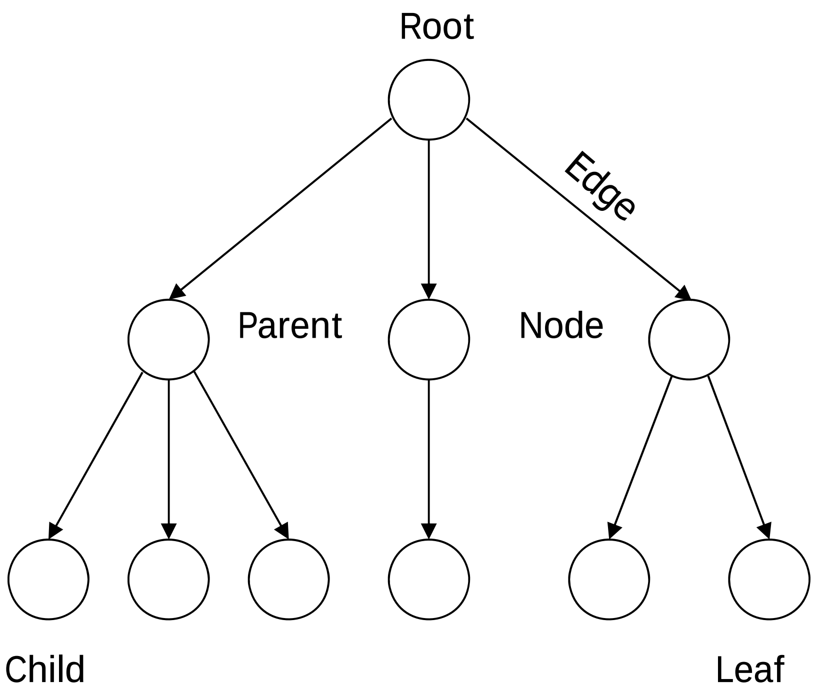
Trees 1

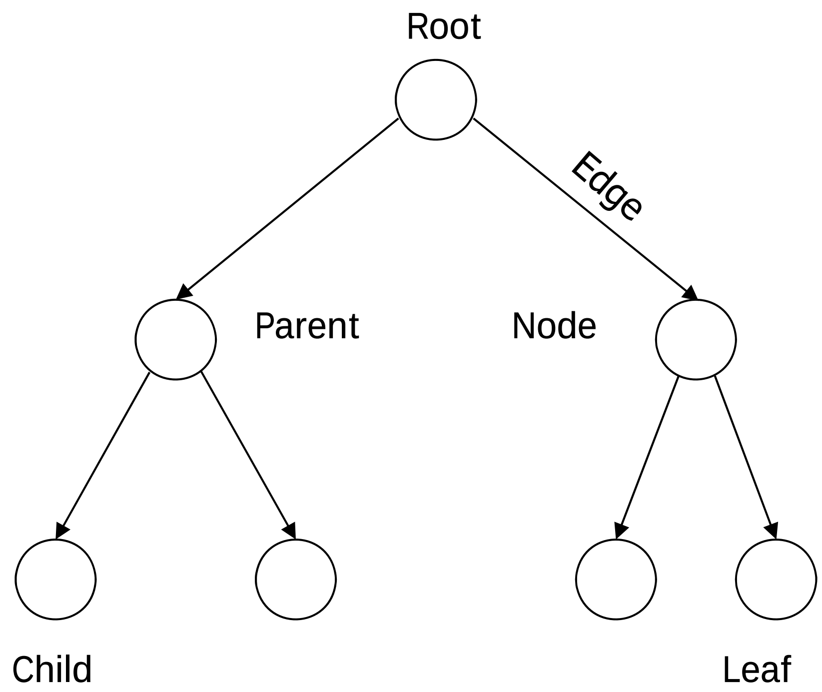
General➔ Binary

General & Binary Trees

- General tree



- Binary Tree



Ordered & Unordered Trees

- A tree is **ORDERED** if the child nodes are considered to be a **SEQUENCE**
- A tree is **UNORDERED** if the child nodes are considered to be a **SET**

Ordered & Unordered Trees

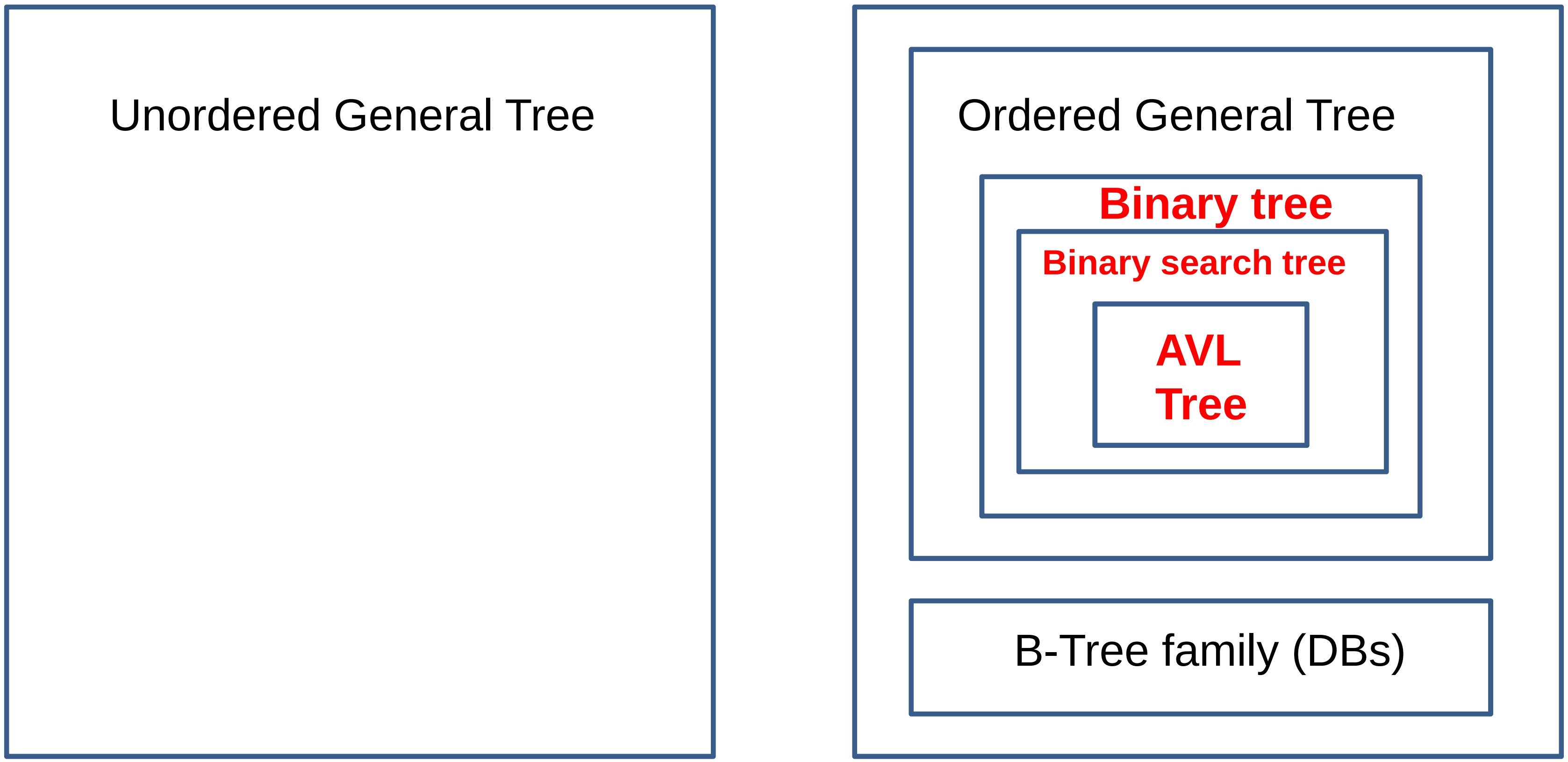
- A **general tree** may thus be **ORDERED** or **UNORDERED**
- A **general tree** with 2 children may be **ORDERED** or **UNORDERED**
- An **ORDERED** **general tree** with 2 children is a **Binary Tree**
- **The children are denoted <u>LEFT</u> & <u>RIGHT</u>**

The Binary Tree Family

- Binary Tree (**BT**) – ordered + 2 children
- Binary Search Tree (**BST**) is **BT** plus
  - value of **left child** < value of the node
  - value of **right child** > value of the node
- **AVL Tree** (Adelson-Velsky Landis)
  - A **BST** where the height difference < 2
  - **| height(LC) – height(RC) | < 2**

General & Binary Trees

Unordered Trees **Ordered Trees**



Properties & Operations

- **General tree**
- **Root**
  - In-degree 0
  - Out-degree **n** (max)
- **Node**
  - In-degree 1
  - Out-degree **n** (max)
- **Leaf Node**
  - In-degree 1
  - Out-degree 0

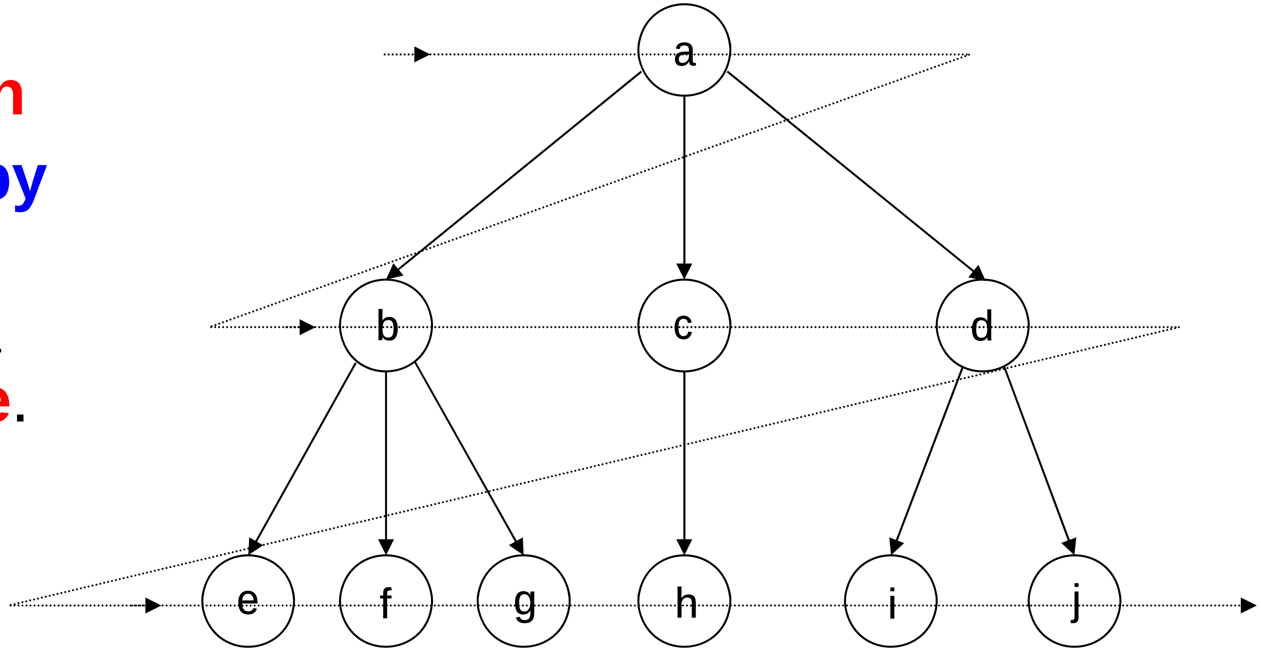
- **Binary Tree**
- **Root**
  - In-degree 0
  - Out-degree **2** (max)
- **Node**
  - In-degree 1
  - Out-degree **2** (max)
- **Leaf Node**
  - In-degree 1
  - Out-degree 0

Properties & Operations

- **ORDERED** (left to right)
  - The children of a node are a **SEQUENCE**
- **UNORDERED**
  - The children of a node are a **SET**
- **Hierarchical** (**parent/child**) organisation
- **Navigation: tree → sequence**
  - Depth-first search (**pre-, in-, post-order; stack**)
  - Breadth-first search (**breadth-first order; queue**)

Tree Traversals

- **Breadth First Search**
- **level by level**
- uses a **Queue**.



a, b, c, d, e, f, g, h, i, j

Definition: General Tree

GT ::= RN $C_1 \ldots C_n$ | empty
RN ::= element

C_1 ::= **GT**
…
C_n ::= **GT**

RN = Root Node
C_i = Child Node

Empty tree; tree with 1 node; tree with n nodes
A collection of nodes and relationships (parent/child)

Definition: Binary Tree

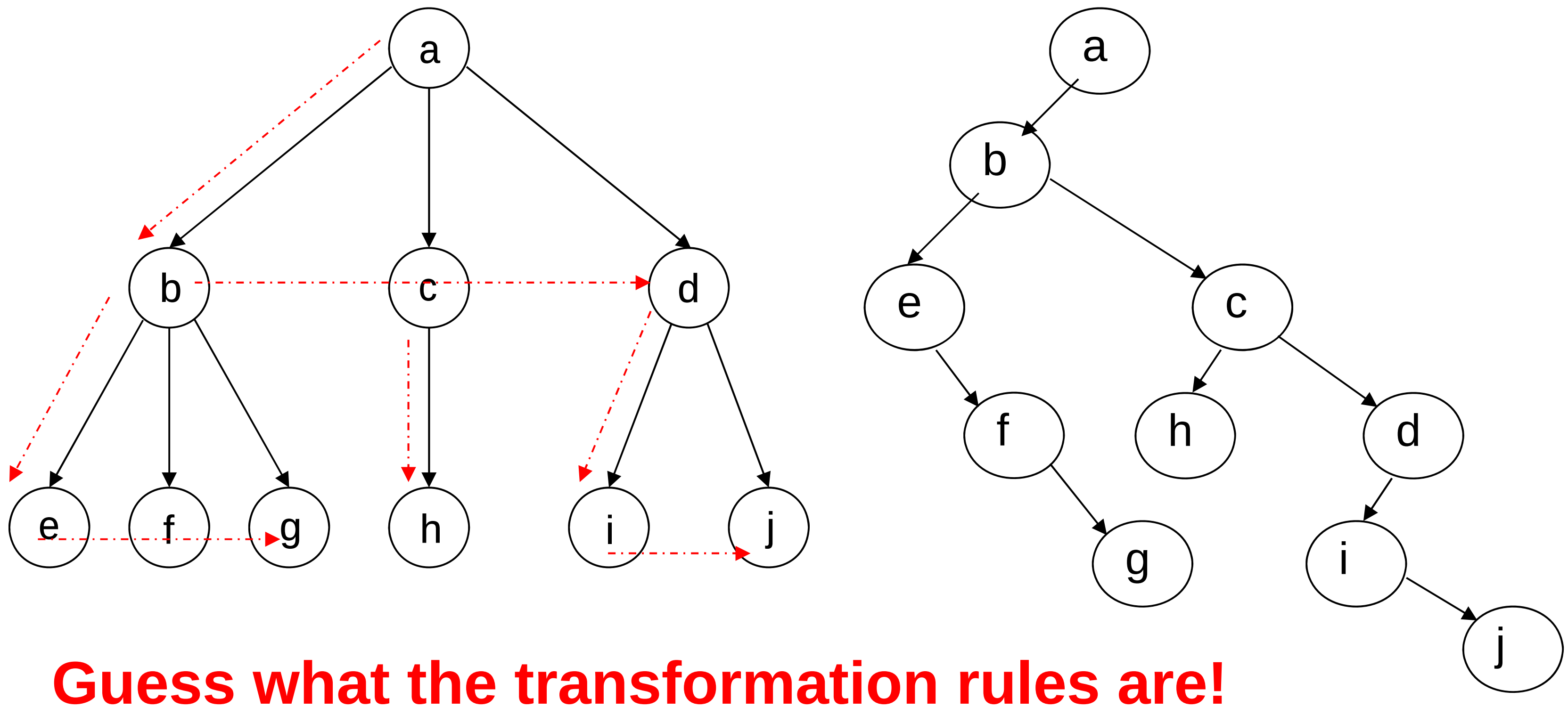
BT ::= LC RN RC | empty
RN ::= element
LC ::= **BT**
RC ::= **BT**

RN = Root Node
LC = left child
RC = right child

➔ **ordered tree (LC, RC) – required for depth-first searches**

Empty tree; tree with 1 node; tree with n nodes
A collection of nodes and relationships (parent/child)

General Tree ➔ Binary Tree



Guess what the transformation rules are!

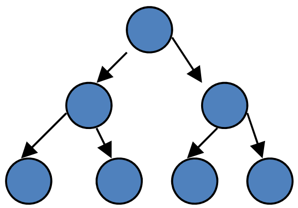
General Tree → Binary Tree

1. The **first child** becomes the **left child** of the parent
2. The **subsequent children** become the **right child** of their **predecessor**

- Example: **a** with children **(b, c, d)**
  - **b** is the **left child** of **a** (rule 1)
  - **c** is the **right child** of **b** (rule 2)
  - **d** is the **right child** of **c** (rule 2)

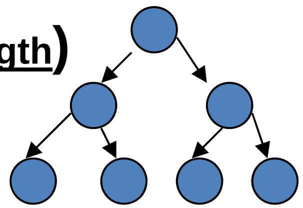
Properties & Operations

- **Height – general tree** (**<u>number nodes / levels</u>**)
  - Height(empty tree) = 0
  - Height(one node) = 1
  - **Height(T) = 1 + max(height(C_1), …,height(C_n))**



- **Height – binary tree** (**<u>number nodes / levels</u>**)
  - Height(empty tree) = 0
  - Height(one node) = 1
  - **Height(BT) = 1 + max(height(LC), height(RC))**
- **Operations on collections apply**
  - Is_empty, add, find, remove, cardinality, …

Height (Depth) revisited



- **Height – general tree** (<u>**number edges / path length**</u>)
  - Height(empty tree) = -1
  - Height(one node) = 0
  - **Height(T) = 0 + max(height(C_1), …,height(C_n))**
- **Height – binary tree** (<u>**number edges / path length**</u>)
  - Height(empty tree) = -1
  - Height(one node) = 0
  - **Height(BT) = 0 + max(height(LC), height(RC))**
- **Operations on collections apply**
  - Is_empty, add, remove, cardinality, …

Caveat Emptor! A Warning

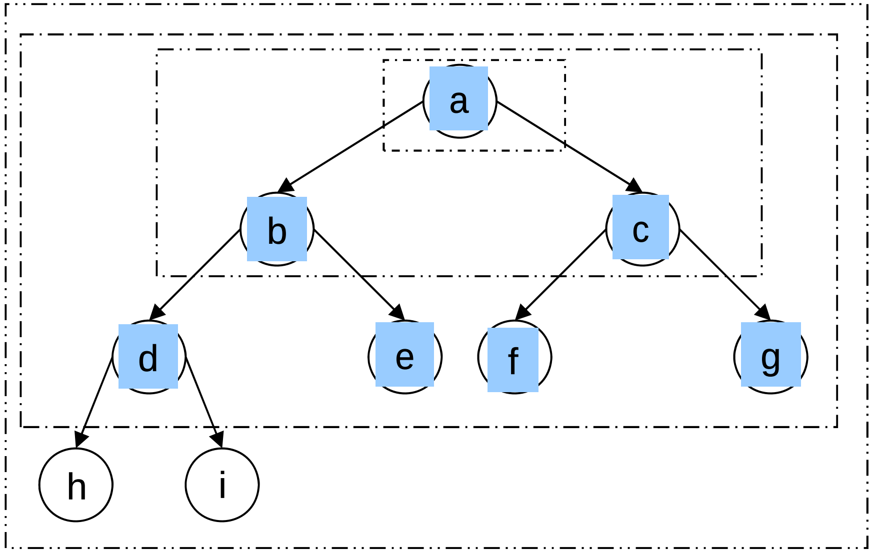
- Be aware of the possibility of different definitions
- Check which definition the article you are reading is in fact using
- This applies also to other structures for example B-Trees (degree)

Binary Tree: properties

FULL: **every node has exactly 2 or 0 children**

PERFECT: BT height **h** with exactly **2^h-1** elements
(NB sometimes called COMPLETE)

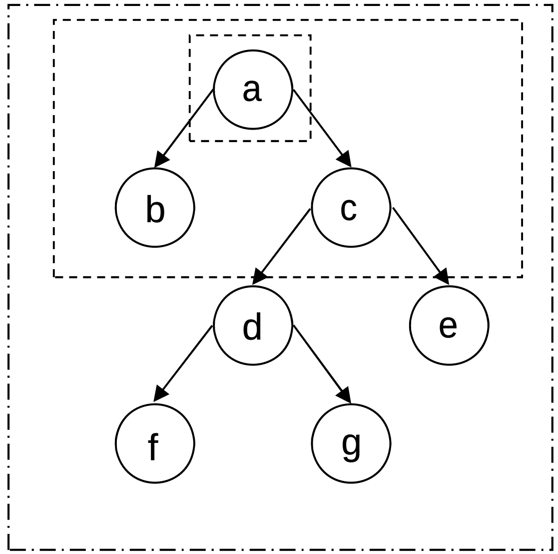
COMPLETE: **perfect** on the **next lowest level** AND the lowest level is filled **from the left**



This allows sequential add to the tree in **breadth-first** position 1 (root), 2, 3, etc. remove is the reverse of this. The BT may be represented as an **array** **(lab1 T2Q)**

Binary Tree: properties

- The number of nodes, ***k,*** in a binary tree, with height ***h***, is defined as **$h \leq k \leq 2^h - 1$**
- **Example 1**
  - **Height = 4, #nodes = 7**
  - **$4 \leq 7 \leq 15$**
- **Example 2**
  - **Height = 2, #nodes = 3**
  - **$2 \leq 3 \leq 3$**



Binary Tree: traversals

- Breadth-first
- Depth-first
  - **pre-order** **NLR** **N = node**
  - **in-order** **LNR** **L = left**
  - **post-order** **LRN** **R = right**
  - **general-order** $\mathbf{N_1}$ **go(L)** $\mathbf{N_2}$ **go(R)** $\mathbf{N_3}$
    - Where 1 = pre-, 2 = in-, 3 = post-order
- Traversals are tree → sequence

Binary Tree: Traversal Algorithms

```
BreadthFirst(T) {
   if T is not Empty {
      Q = Empty;
      Q = AddQ(Q, T);
      while(Q != Empty) {
         p = front(Q); Q = deQ(Q);
         process(Root(p));
         if(Left(p)  != Empty) Q = AddQ(Q, Left(p));
         if(Right(p) != Empty) Q = AddQ(Q, Right(p));
      }
   }
}
```

Binary Tree: Traversal Algorithms

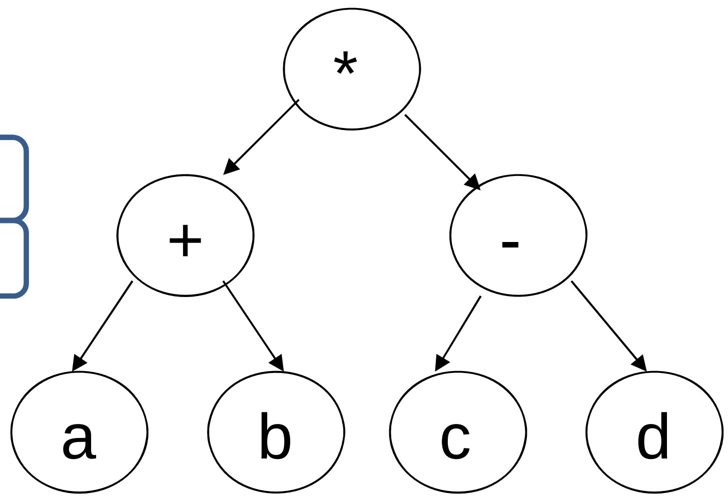
```
PreOrder(T) {
if !is_Empty(T){process(Root(T)); PreOrder(Left(T)); PreOrder(Right(T));}
}
```

```
InOrder(T) {
if !is_Empty(T){ InOrder(Left(T)); process(Root(T)); InOrder(Right(T));}
}
```

```
PostOrder(T) {
if !is_Empty(T){ PostOrder(Left(T)); PostOrder(Right(T)); process(Root(T));}
}
```

Binary Tree: arithmetic expressions

- **Arithmetic expressions**
  - **(a+b) * (c-d)**
  - Pre ***+ab-cd**
  - In - **a+b*c-d** **!!**
  - In - **(a+b)*(c-d)**
  - Post **ab+cd-***
- **Note “(“ & “)” have to be reinserted to re-create the infix!**



- Infix ➔ pre-/post-fix

Exercise: infix → postfix

(a+b) * (c-d)

stack: (o/p:

stack: (o/p: a

stack: (+ o/p: a

stack: (+ o/p: ab

stack: o/p: ab+

stack: * o/p: ab+

stack: * (o/p: ab+

stack: * (o/p: ab+c

stack: * (- o/p: ab+c

stack: * (- o/p: ab+cd

stack: * o/p:ab+cd-

stack: o/p: ab+cd-*

Operator	(: Push
	): Pop to (
*,+	stack / pop

but note precedence!

a+b*c → abc*+

a*b+c → ab*c+

NB: Operand: Output (o/p)

o/p = output

Exercise: postfix ➔ tree

ab+cd-*