

Trees

Let us now present the concept of **tree**. A tree is either

- the empty set
- or a tuple made of a **root** and other trees, called **subtrees**.

This is a **recursive definition** because the object (here, the tree) is defined by case and by grouping objects of the same kind (here, the subtrees).

A root could be further refined as containing some specific information.

It is usual to call **nodes** the root of a given tree and the roots of all its subtrees, *transitively*.

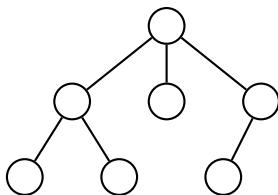
A node without non-empty subtrees is called a **leaf**.

Trees (cont)

If we consider trees as relationships between nodes, it is usual to call a root the **parent** of the roots of its direct subtrees (i.e., the ones immediately in the tuple). Conversely, these roots are **sons** of their parent (they are ordered).

It is also common to call subtree any tree included in it according to the subset relationship (otherwise we speak of *direct* subtrees).

Trees are often represented in a top-down way, the root being at the top of the page, nodes as circles and the relationship between nodes as **edges**. For instance:



Trees (cont)

The **depth** of a node is the length of the path from the root to it (note this path is unique). Thus the depth of the root is 0 and the depth of the empty tree is undefined.

The **height** of a tree is the maximal depth of its nodes. For example, the height of the tree in the previous page is 2.

A **level** in a tree is the set of all nodes with a given depth. Hence it is possible to define level 0, level 1 etc. (may be empty).

Trees/Traversals

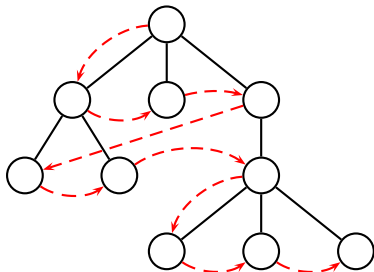
Given a tree, we can traverse it in many ways, but we must start from the root since we do not have any other node at this level.

There are two main kind of traversals:

- **Breadth-first** traversal consists in walking the tree by increasing levels: first the root (level 0), the sons of the root (level 1), then the sons of the sons (level 2) etc.
- **Depth-first** traversal consists in walking the tree by reaching the leaves as soon as possible.

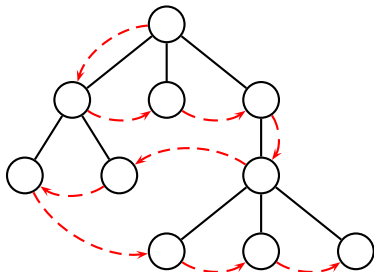
In both cases, we are finished when all the leaves have been encountered.

Trees/Traversals/Breadth-first

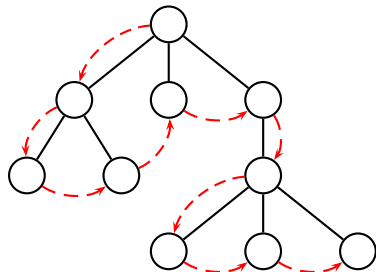


This is a **left to right** traversal.

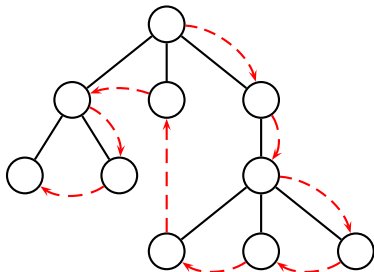
Many others are possible, like choosing randomly the next node of the next level.



Trees/Traversals/Depth-first



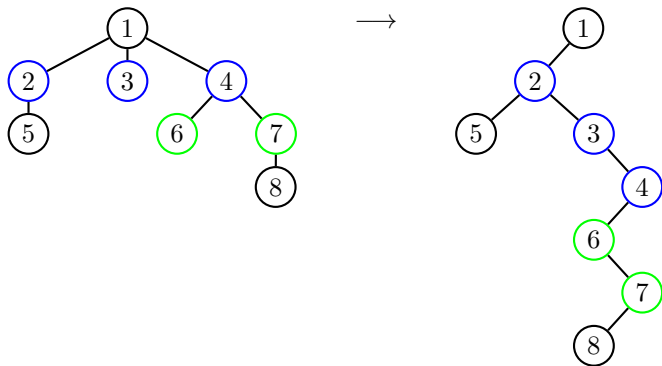
This is a **left to right** traversal.



This is a **right to left** traversal.

Trees/Binary trees

Let us consider for a while binary trees only. Indeed, we do not lose any generality with this apparent restriction because it is always possible to map any tree to a binary tree in a unique way. One method is the **left child/right sibling**:




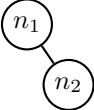
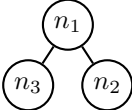
Trees/Formal definition

Let us formalise the concept of tree.

- Let us note `EMPTY` the empty tree. If the context is ambiguous, one can qualify the notation with the name of the data structure, for example, write `TREE.EMPTY` in order to make clear that we do not refer to empty stacks or empty queues.
- Let us note `JOIN(r, t_1, t_2)` the tree whose root is r , left subtree is t_1 and right subtree is t_2 .

Trees/Examples

Let us show some examples of tree construction:

EMPTY	\emptyset
$\text{JOIN}(n_1, \text{EMPTY}, \text{EMPTY})$	
$\text{JOIN}(n_1, \text{EMPTY}, \text{JOIN}(n_2, \text{EMPTY}, \text{EMPTY}))$	
$\text{JOIN}(n_1, \text{JOIN}(n_3, \text{EMPTY}, \text{EMPTY}), \text{JOIN}(n_2, \text{EMPTY}, \text{EMPTY}))$	

Trees/Basic projections

Let us define the basic operations consisting in extracting the root, the left subtree and the right subtree of a binary tree.

$$\text{ROOT}(\text{JOIN}(r, t_1, t_2)) \rightarrow r$$

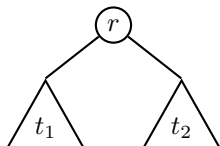
$$\text{LEFT}(\text{JOIN}(r, t_1, t_2)) \rightarrow t_1$$

$$\text{RIGHT}(\text{JOIN}(r, t_1, t_2)) \rightarrow t_2$$

Note that these three operations are not defined on empty trees. In a programming language, the case of the empty tree would be handled explicitly, either by returning an error code (C-like convention) or throwing an exception (C++ like).

We also could specify the error situation in the rewrite system, as we did for the division by zero, but we prefer not, in order to focus on the concept, not the errors.

Trees/Left to right traversals



A depth-first traversal from left to right first visits node r , then the left subtree t_1 and finally the right subtree t_2 . But if we want to keep track of the visited nodes, we have several ways.

- We record r , then nodes of t_1 and nodes of t_2 : this is **left prefix traversal**;
- we record nodes of t_1 , then r and nodes of t_2 : this is a **left infix traversal**;
- we record nodes of t_1 , then nodes of t_2 and r : this is a **left postfix traversal**.

Trees/Left-to-right prefix traversal

In order to record the traversed nodes, we need an additional structure: a stack.

The rewrite system corresponding to a left-to-right prefix traversal is

$$\text{LPREF}(\text{EMPTY}) \rightarrow \text{EMPTY}$$

$$\text{LPREF}(\overline{\text{JOIN}(e, t_1, t_2)}) \rightarrow \text{PUSH}(\bar{e}, \text{APPEND}(\text{LPREF}(\overline{t_1}), \text{LPREF}(\overline{t_2})))$$

Trees/Left-to-right postfix traversal

The rewrite system corresponding to a left-to-right postfix traversal is

$$\text{LPOST}(\text{EMPTY}) \rightarrow \text{EMPTY}$$

$$\begin{aligned} \text{LPOST}(\overline{\text{JOIN}(e, t_1, t_2)}) \rightarrow \\ \text{APPEND}(\text{LPOST}(\overline{t_1}), \text{APPEND}(\text{LPOST}(\overline{t_2}), \text{PUSH}(\overline{e}, \text{EMPTY}))) \end{aligned}$$

Trees/Left-to-right infix traversal

The rewrite system corresponding to a left-to-right infix traversal is

$$\text{LINF}(\text{EMPTY}) \rightarrow \text{EMPTY}$$

$$\text{LINF}(\text{JOIN}(e, t_1, t_2)) \rightarrow \text{APPEND}(\text{LINF}(t_1), \text{PUSH}(e, \text{LINF}(t_2)))$$