## Trees: Basic terminology (1)

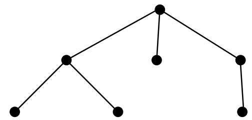
Examples for tree structures:

+ genealogic trees
  (successors of a person)
+ hierarchical classification systems in science and engineering
+ hierarchical organization diagrams
  (company: departments, divisions, groups, employees)
+ structured documents
  (book: chapters, sections, subsections, paragraphs, ...)
+ expression trees



## Trees: Basic terminology (2)

*Tree* = set of nodes and edges, $T = (V, E)$.



Nodes $v \in V$ store data items in a *parent-child* relationship.

A parent-child relation between nodes $u$ and $v$ is shown as a *directed edge* $(u, v) \in E$, from $u$ to $v$. $E \subset V \times V$

Each node in a tree $T$ has at most one parent node:
$\forall v \in V : |\{(u, v) \in E : u \in V\}| \leq 1$

There is exactly one node that has no parent: the *root* of $T$.

The *degree* of a node $v \in V$ is the number of its children: $|\{(v, w) \in E : w \in V\}|$

A node that has no children is called a *leaf node*.



## Trees: Basic terminology (3)

Formal (inductive) definition of a *tree*:

All trees are characterized by the following construction rules:

- A single node, with no edges, is a tree.
- Let $T_1, ..., T_k$ ($k \geq 1$) be trees with no nodes in common.
  Let $r_i$ denote the root of $T_i$, for $1 \leq i \leq k$.
  Let $r$ be a new node.
  Then there is a tree $T$ consisting of all nodes and edges of $T_1, ..., T_k$, the new node $r$, and the edges $(r, r_1)$, ..., $(r, r_k)$.

Remarks on the second rule:

$r$ is the root of the new tree $T$.

$r_1, ..., r_k$ are *children* of $r$ and *siblings* of each other.

$T_1$,...,$T_k$ are the *subtrees* of $T$.

$k$ is the *degree* of $r$.



## Trees: Basic terminology (4)

*path* $\pi = (v_1, v_2, ..., v_l)$ in $T = (V, E)$ from $v_1$ to $v_l$ with length $l - 1$
if $v_i \in V \; \forall i, \; 1 \leq i \leq l$, and $(v_i, v_{i+1}) \in E \; \forall i, \; 1 \leq i < l$

*ancestors* of a node $v \in V$: $\{u \in V : \exists$ path from $u$ to $v$ in $T\}$

*successors* of a node $v \in V$: $\{w \in V : \exists$ path from $v$ to $w$ in $T\}$

*depth* $d(v)$ of a node $v \in V$
length of longest path from the root to $v$

*height* $h(v)$ of a node $v \in V$
length of longest path from $v$ to a successor of $v$

*height* $h(T)$ of tree $T$ = height of the root of $T$

## Special kinds of trees

Ordered tree: linear order among the children of each node

Binary tree: ordered tree with degree $\leq 2$ for each node
$\Rightarrow$ left child, right child

Empty binary tree ($\Lambda$): binary tree with no nodes

Full binary tree: nonempty; degree is either 0 or 2 for each node
Fact: number of leaves = 1 + number of interior nodes (proof by induction)

Perfect binary tree: full, all leaves have the same depth
Fact: number of leaves = $2^h$ for a perfect binary tree of height $h$
(proof by induction on $h$)

Complete binary tree: approximation to perfect for $2^h \leq n < 2^{h+1} - 1$

Forest: finite set of trees, i.e., multiple roots possible



## ADT Tree (1)

Domain: tree nodes, maybe associated with additional information

Operations on a single tree node $v$:

$Parent(v)$ returns parent of $v$, or $\Lambda$ if $v$ root
$Children(v)$ returns set of children of $v$, or $\Lambda$ if $v$ leaf
$FirstChild(v)$ returns first child of $v$, or $\Lambda$ if $v$ leaf
$LeftChild(v)$, $RightChild(v)$ returns left / right child of $v$, or $\Lambda$ if not existing
$RightSibling(v)$ returns right sibling of $v$, or $\Lambda$ if $v$ is a rightmost child
$LeftSibling(v)$ returns left sibling of $v$, or $\Lambda$ if $v$ is a leftmost child
$IsLeaf(v)$ returns true iff $v$ is a leaf



## ADT Tree (2)

Operations on an entire tree $T$:

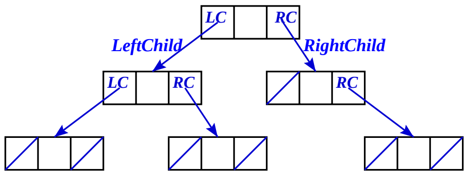
$Size(T)$ returns number of nodes of $T$
$Root(T)$ returns root node of $T$
$IsRoot(v,T)$ returns true iff $v$ is root of $T$
$Depth(v,T)$ returns depth of $v$ in $T$
$Height(v,T)$ returns height of $v$ in $T$
$Depth(T)$ returns length of longest path in $T$
$Height(T)$ returns height of the root of $T$



## Tree representations (1): using pointers

Type **Tnode** denotes a **pointer** to a structure storing node information:

**record node_record**
*nchilds*: **integer**
*child*: **table**<**Tnode**> [1..*nchilds*]
*info*: **infotype**




For binary trees:
2 pointers per node, $LC$ and $RC$

Alternatively, the pointers to a node's children can be stored in a linked list.

If required, a "backward" pointer to the parent node can be added.

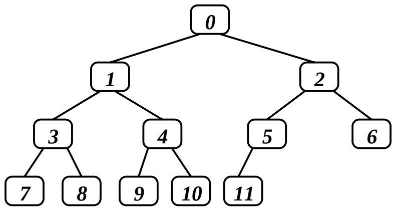
Insertion and deletion take constant time.

## Tree representations (2): array indexing

For a complete binary tree holds:

*There is exactly one complete binary tree with $n$ nodes.*

Implicit representation of edges: Numbering of nodes $\rightarrow$ index positions





$LeftChild(i)$: $2i+1$
(none if $2i+1 \geq n$)
$RightChild(i)$: $2i+2$
(none if $2i+1 \geq n$)
$IsLeaf(i)$: $2i+1 > n$
$LeftSibling(i)$: $i-1$
(none if $i=0$ or $i$ odd)
$RightSibling(i)$: $i+1$
(none if $i=n-1$ or $i$ even)
$Parent(i)$: $\lfloor (i-1)/2 \rfloor$ (none if $i=0$)
$Depth(i)$: $\lfloor \log_2(i+1) \rfloor$
$Height(i)$: $\lfloor \log_2((n+1)/(i+1)) \rfloor$



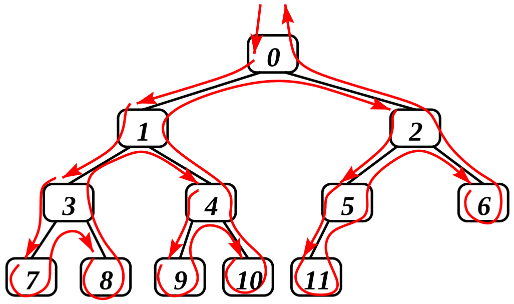
## Tree traversals (1)

Regard a tree $T$ as a building:
nodes as rooms, edges as doors, root as entry

How to explore an unknown (acyclic) labyrinth and get out again?
Proceed by always keeping a wall to the right!

Generic tree traversal routine:

**procedure** *visit* ( **node** $v$ )
{ explore subtree rooted at $v$ }
**for all** $u \in Children(v)$ **do**
$visit(u)$




Call *visit*( $Root(T)$ ):
each node in $T$ will be visited exactly once (proof by induction)



## Tree traversals (2)

**procedure** *preorder_visit* ( **node** $v$ )
**output** $v$ { before any of the subtree nodes are output }
**for all** $u \in Children(v)$ **do**
*preorder_visit*$(u)$

**procedure** *postorder_visit* ( **node** $v$ )
**for all** $u \in Children(v)$ **do**
*postorder_visit*$(u)$
**output** $v$ { after all of the subtree nodes have been output }

**procedure** *inorder_visit* ( **node** $v$ ) { only for binary trees }
*inorder_visit*$(LC(v))$
**output** $v$
*inorder_visit*$(RC(v))$



## Implementing Sets and Dictionaries as Binary Search Trees

A *binary search tree* (BST) is a binary tree such that:

- Information associated with a node includes a *key*,
  $\rightarrow$ linear ordering of nodes determined by keys.
- The key of each node is:
  greater than the keys of all left descendants, and
  smaller than the keys of all right descendants.