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 \( v \). \( E \subseteq V \times V \)

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 \( T_1, \ldots, T_k \) \((k \geq 1)\) 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 \( T_1, \ldots, T_k \),
  the new node \( r \), and the edges \((r, r_1), \ldots, (r, r_k)\).

Remarks on the second rule:
- \( r \) is the root of the new tree \( T \).
- \( r_1, \ldots, r_k \) are children of \( r \) and siblings of each other.
- \( T_1, \ldots, T_k \) are the subtrees of \( T \).
- \( k \) is the degree of \( r \).

Trees: Basic terminology (4)

path \( \pi = (v_1, v_2, \ldots, v_l) \) 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 \( v \in V \): \( \{u \in V : \exists \text{ path from } u \text{ to } v \text{ in } T\} \)

successors of a node \( v \in V \): \( \{w \in V : \exists \text{ path from } v \text{ to } w \text{ in } T\} \)

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

⇒ 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 \( v \):

- \( \text{Parent}(v) \) returns parent of \( v \), or \( \Lambda \) if \( v \) root
- \( \text{Children}(v) \) returns set of children of \( v \), or \( \Lambda \) if \( v \) leaf
- \( \text{FirstChild}(v) \) returns first child of \( v \), or \( \Lambda \) if \( v \) leaf
- \( \text{RightSibling}(v) \) returns right sibling of \( v \), or \( \Lambda \) if not existing
- \( \text{LeftSibling}(v) \) returns left sibling of \( v \), or \( \Lambda \) if \( v \) is a leftmost child
- \( \text{IsLeaf}(v) \) returns true iff \( v \) is a leaf

ADT Tree (2)

Operations on an entire tree \( T \):

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

Tree representations (1): using pointers

Type \( \text{Tnode} \) denotes a pointer to a structure storing node information:

\[
\begin{align*}
\text{record} & \quad \text{node_record} \\
& \quad \text{nchilds: integer} \\
& \quad \text{child: table} <\text{Tnode}> [1..\text{nchilds}] \\
& \quad \text{info: infotype}
\end{align*}
\]

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 ➔ index positions

\[
\begin{align*}
\text{LeftChild}(i) : & \quad 2i + 1  \\
& \text{ (none if } 2i + 1 \geq n) \\
\text{RightChild}(i) : & \quad 2i + 2  \\
& \text{ (none if } 2i + 1 \geq n) \\
\text{IsLeaf}(i) : & \quad 2i + 1 > n  \\
\text{LeftSibling}(i) : & \quad i  \\
& \text{ (none if } i = 0 \text{ or } i \text{ odd) } \\
\text{RightSibling}(i) : & \quad i + 1  \\
& \text{ (none if } i = n - 1 \text{ or } i \text{ even) } \\
\text{Parent}(i) : & \quad \lfloor (i - 1)/2 \rfloor  \\
& \text{ (none if } i = 0) \\
\text{Depth}(i) : & \quad \lceil \log_2(i + 1) \rceil  \\
\text{Height}(i) : & \quad \lceil \log_2((n + 1)/(i + 1)) \rceil
\end{align*}
\]

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:

\[
\begin{align*}
\text{procedure } & \text{visit (node } v) \{ \text{ before any of the subtree nodes are output } \} \\
& \text{for all } u \in \text{Children}(v) \text{ do } \\
& \quad \text{visit}(u) \\
\text{Call visit( Root}(T) ) : \\
& \quad \text{each node in } T \text{ will be visited exactly once (proof by induction)}
\end{align*}
\]

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, ➔ 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.