**MCS-375: Algorithms: Analysis and Design** Handout #DS1
**San Skulrattanakulchai**
Gustavus Adolphus College Sep 16, 2013

# Binary Trees

**CLRS: Ch 10.4, Appendix B.5.2–B.5.3**

A binary tree is either empty or nonempty, but not both. A nonempty binary tree $B$ consists of three things:

(i) a node $r$ called *root*,

(ii) a left binary tree $L$ called *left subtree*, and

(iii) a right binary tree $R$ called *right subtree*.

We write $B = (r, L, R)$ to denote the nonempty binary tree $B$. We assume no node is the root of more than one binary tree, and no binary tree is the subtree of itself.

**Representing Binary Trees** Each node in a binary tree $T$ contains a `key` field, `left` and `right` fields pointing to the left subtree and the right subtree, respectively. If traversing up the tree is desired, the node should have another field `p` pointing to the parent node. The binary tree $T$ needs only contain the root node in its `root` field.
An empty binary tree is represented by a special value called `NIL`. For technical reason, `NIL` may be implemented as a real node.

**Exercise** (cf CLRS 12.1-4) Give recursive algorithms to visit the nodes of a binary tree in *preorder*, *inorder*, and *postorder*. What are the running times of your algorithms?

A node $x$ is *in* a binary tree $B = (r, L, R)$ if $x = r$, or $x$ is in $L$, or $x$ is in $R$. The phrase "$B$ *contains* $x$" means "$x$ is in $B$."

A node $y$ is the *right child* of a node $x$ if there is a binary tree $B = (x, R, L)$ with $y$ being the root of $R$.

**Exercise** Define *left child*, *child*, *sibling*, *parent*, *grandparent*, *grandchild*, *uncle*, and *nephew*.

A node $x$ is an *ancestor* of a node $y$ if there exists some binary tree $B = (x, R, L)$ containing $y$. Node $x$ is a *proper ancestor* of node $y$ if $x \neq y$ and $x$ is an ancestor of $y$.

**Exercise** Define *descendant* and *proper descendant*.

Let $x$ be a node in some binary tree $B$. Node $x$ is called *leaf* if it is the parent of no other node; it is called an *internal node* otherwise.

A *path* is a sequence $v_0, v_1, v_2, \ldots, v_k$ of nodes such that for each $0 \leq i < k$, node $v_i$ is either the parent or a child of node $v_{i+1}$. A *simple path* has no repeated nodes.

An *edge* is a pair $\{x, y\}$ of nodes such that $x$ is either the parent or a child of $y$. The *length* of a path is the number of edges it contains.

The *depth* of a node $x$ in a binary tree $B = (r, L, R)$, denoted $\text{depth}(x, B)$, is defined as

$$\text{depth}(x, B) = \begin{cases} 0 & \text{if } x = r \\ 1 + \text{depth}(x, L) & \text{if } x \text{ is in } L \\ 1 + \text{depth}(x, R) & \text{if } x \text{ is in } R. \end{cases}$$

The height of a binary tree $B$, denoted $\text{height}(B)$, is defined as

$$\text{height}(B) = \max\{\, \text{depth}(x, B) : x \text{ is in } B \,\}.$$

The *level* $\ell$ of $B$ consists of all nodes in $B$ at depth $\ell$.

A node in a binary tree is *full* if it has two children. A binary tree is *full* if each of its internal nodes is full. A binary tree is *complete* if it is full and all leaves have the same depth.

**Exercise** Write programs to

1. find out if a node $x$ is in binary tree $B$,
2. compute the number of nodes in $B$,
3. compute the number of leaves in $B$,
4. find out if a binary tree $B$ has a edge joining node $x$ to node $y$,
5. compute the depth of a node $x$ in a binary tree $B$,
6. compute the height of a binary tree $B$.

What are the running times of your programs?

**Full Binary Tree Lemma (CLRS B.5-3)** The number of internal nodes in any nonempty full binary tree is one less than the number of leaves.

*Proof.* By induction, or counting the # edges in two different ways. □

A complete binary tree of height $h$ has $1 + 2 + 2^2 + \cdots + 2^h = 2^{h+1} - 1$ nodes.

**Binary Tree Height Lemma (CLRS B.5-4)** A binary tree $B$ of $n$ nodes has height$(B) \geq \lg(n + 1) - 1$.

*Proof.* This follows from $n \leq 2^{h+1} - 1$. □

For each integer $k \geq 2$, a *$k$-ary tree* is defined similarly to a binary tree.
A $k$-ary tree node can be represented in a similar manner to a binary tree node. Instead of having two pointers `left` and `right` pointing to the left and right subtrees, it has a $k$-array pointing to the $k$ subtrees.

A rooted tree $T$ is either empty or consists of a node called the *root* of $T$, and zero or more rooted trees called the *subtrees* of the root.

An *ordered tree* is a rooted tree where the subtrees are presented in a specific order.
An ordered tree can be represented efficiently by making each node have 2 fields: a `left-child` and a `right-sibling`.

All notions on binary trees, e.g., child, ancestor, depth, edge, tree traversal, etc., extend to rooted trees.