Trees

Textbook: Chapter 5, Section 5.5

Tree traversals: Notes on Graphs and Trees by Cusack

Textbook: Chapter 13, Section 13.1

CSCE310: Data Structures and Algorithms

`www.cse.unl.edu/~choueiry/S01-310/`

Berthe Y. Choueiry (Shu-we-ri)

Ferguson Hall, Room 104

`choueiry@cse.unl.edu`, Tel: (402)472-5444

Free tree

A connected, acyclic, undirected graph $\longrightarrow$ Tree

A possibly disconnected, acyclic, undirected graph $\longrightarrow$ Forest

Let $G = (V, E)$ be an undirected graph. The following statements are equivalent.

1. G is a free tree.
2. Any two vertices in G are connected by a unique simple path.
3. G is connected, but if any edge is removed from E, the resulting graph is disconnected.
4. G is connected, and $|E| = |V| - 1$.
5. G is acyclic, and $|E| = |V| - 1$.
6. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle.

(1) $\rightarrow$ (2) $\left\{ \begin{array}{l} \text{(1) } G \text{ is a free tree } (\equiv \text{ connected, acyclic, undirected graph}) \\ \text{(2) Any two vertices are connected by a unique simple path} \end{array} \right.$

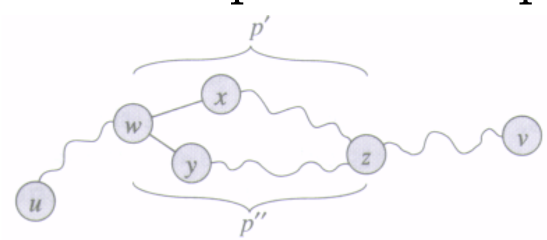
G connected $\rightarrow$ any two vertices are connected by <u>at least</u> one simple path, prove this path is unique by contradiction

Consider u and v, two vertices linked by 2 simple paths p_1 and p_2.

Let w (resp. z) the vertex where p_1 & p_2 converge (resp. diverge).

Let p' (p'') the subpath of p_1 (p_2) from w to z through x (y).

p' and p'' share no vertices except their endpoints



The path obtained by concatenating p' and reverse of p'' is a cycle

The tree is thus cyclic $\Rightarrow$ Contradiction!

There can be at most one path between any two vertices

(2) → (3)

$$\textbf{(2)} \rightarrow \textbf{(3)} \begin{cases} \text{(2) Any two vertices are connected} \\ \quad \text{by a unique simple path} \\ \text{(3) } G \text{ is connected, but if any edge is removed from} \\ \quad E\text{, the resulting graph is disconnected} \end{cases}$$

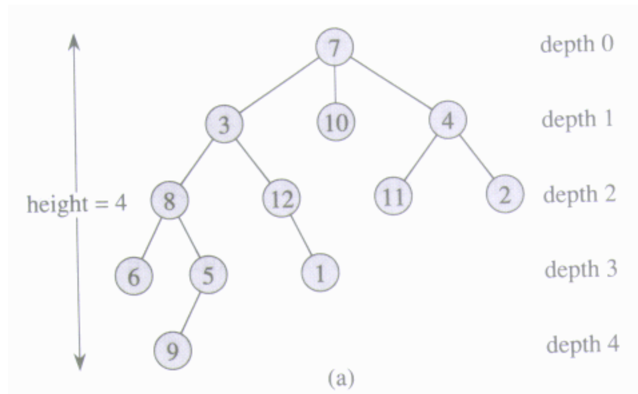
Let (u, v) be any edge in E

This edge is a path from u to v $\Rightarrow$ it must be the unique simple path from u to v

Remove it, and G will be disconnected

Check textbook for: (3) → (4), (4) → (5), (5) → (6), and (6) → (1).

Rooted tree: is a free tree T with a **root** r (distinguished node)



(a)

Ancestor of a node x: A node y on the unique path from x to the root

Descendant of y: any node whose ancestor is y

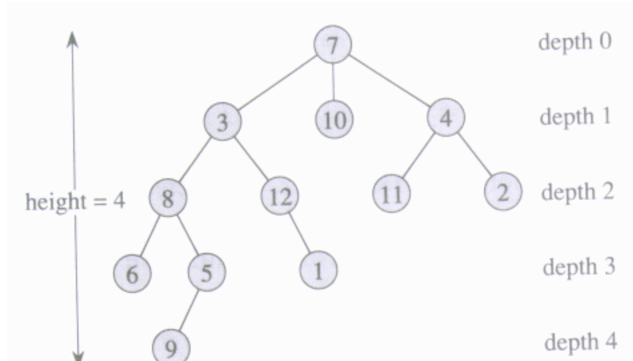
Every node is descendant and ancestor of itself

Proper ancestor: If y is an ancestor of x and $y \neq x$

Proper descendant: If x is a descendant of y and $x \neq y$

Subtree rooted at x: subtree induced by descendants of x, rooted at x

Rooted tree (II)



(a)

Parent of x: y such that (y, x) is the last edge on path from r to x. Only, r has no parent.

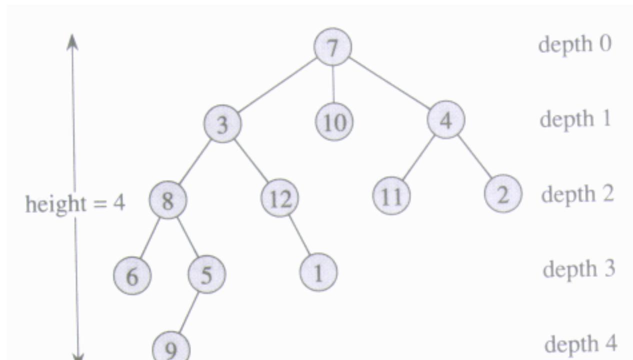
Child of y: x such that (y, x) is the last edge on path from r to x

Siblings: Two nodes with same parents

Leaf, external node: a node with no children

Internal node: nonleaf node

Rooted tree (III)



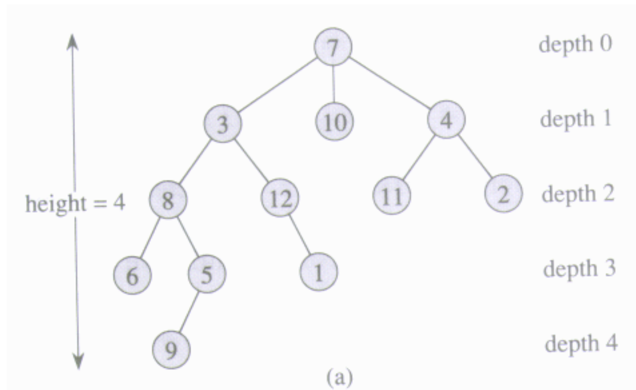
(a)

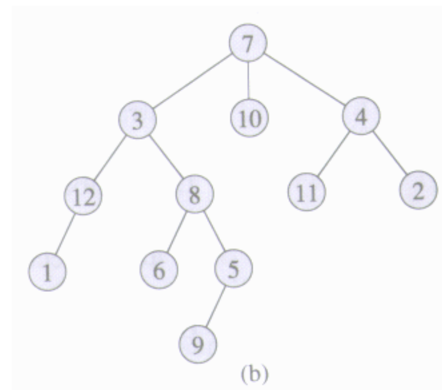
Degree of x: number of children of x

Depth of x in T: length of path from r to x

Height of T: largest depth of any node x in T

Rooted tree (IV)





Ordered tree: Children of <u>each</u> node are ordered (1^{st} child, 2^{nd} child, etc.)

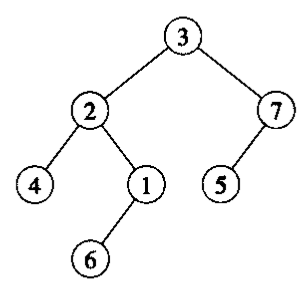
(a) and (b) are different if considered as <u>**ordered**</u> rooted trees

(a) and (b) are same if considered a rooted trees

Binary tree T *(recursive definition)*

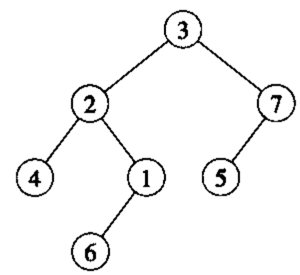
is a structure defined on a finite set of nodes that either

- contains no nodes, or
- is comprised of 3 disjoint sets of nodes
  1. a **root** node (**empty tree**, **null tree**, denoted `Nil`)
  2. a binary tree, called its **left subtree**
  3. a binary tree, called its **right subtree**



(a)

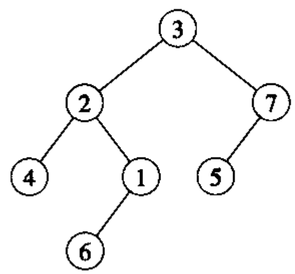
Binary tree T (II)



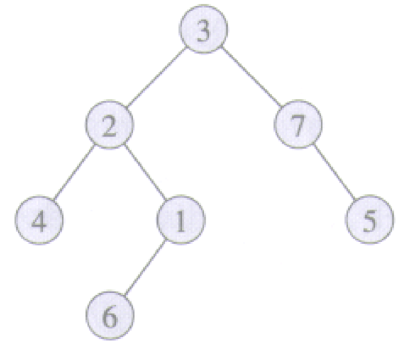
(a)

- If left subtree non empty, its root is the **left child** of root of T
- If right subtree non empty, its root is the **right child** of root of T
- If subtree is the null tree `Nil`, we say child is **absent, missing**

Binary tree T (III)



(a)



FALSE: Binary is an ordered tree in which each node has degree at most 2.

It matters to know the position of an only child: left or right?

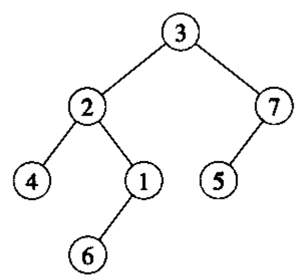
(a) and (b) are the same tree

(a) and (b) are the same ordered tree

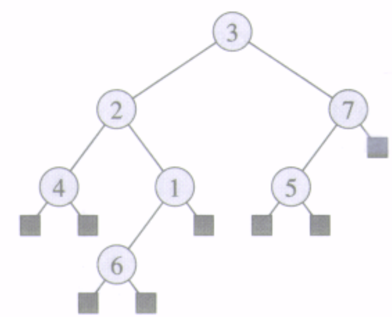
(a) and (b) are **not** the some binary tree

Positioning information

replace each missing child with a node with no children, drawn as a square



(a)



(c)

Result: **full binary tree**, each node $\begin{cases} \text{is either a leaf, or} \\ \text{has a degree 2, exactly} \end{cases}$

Order of children preserves position information

Positional tree

(generalize for k children)

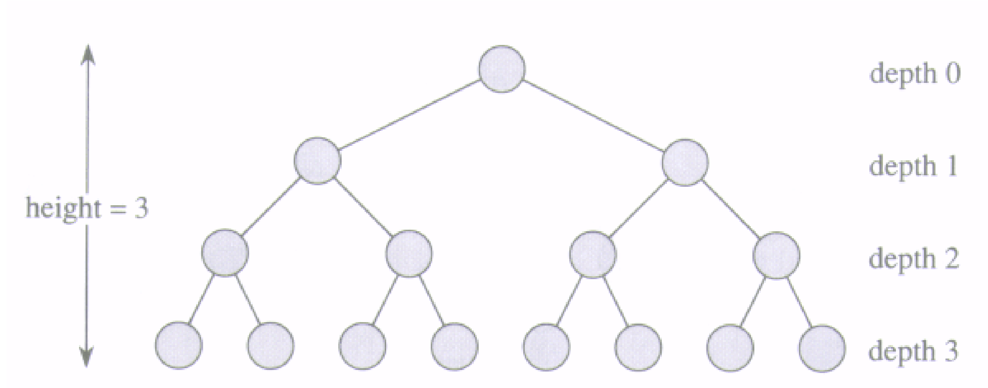
- Children of a node are labeled with distinct positive integers.
- i^{th} child missing if no child is labeled with i

k-ary tree: positional tree with children with labels $> k$ are missing

Binary-tree: is a k-ary tree with $k = 2$

Complete k-ary tree: $\begin{cases} \text{all leaves have the same depth, and} \\ \text{all internal nodes have degree } k \end{cases}$

Complete k-ary tree



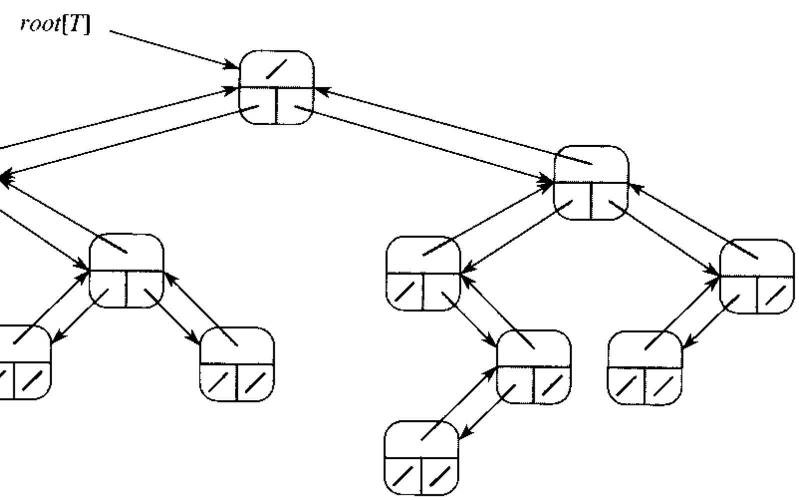
- Number of **<u>leaves</u>** at depth h is .....
- The **<u>height</u>** of a k-ary complete tree with n leaves is ...
- The number of **<u>internal</u>** nodes is:
  $1 + k + k^2 + \ldots + k^{h-1} = \sum_{i=0}^{h-1} k^i = \frac{k^h - 1}{k-1}$
- A complete binary tree has $2^h - 1$ internal nodes.
- A complete binary tree has $2^{(}h + 1) - 1$ nodes.

Binary tree representation as (doubly) linked lists

(see Section 11.4)

Node in T represented

by object with fields: $\begin{cases} key \\ p: \text{ parent}(optional) \\ left: \text{ left child} \\ right: \text{ right child} \end{cases}$



Binary Tree Traversals

Courtesy of C. Cusack

- When we visit each node in the tree exactly once, we say we have **Traversed** the tree.
- A full traversal produces a linear order of the information in a tree.
- There are several ways to traverse a tree.
  1. **Preorder**: visit a node, then traverse its left subtree, and then traverse its right subtree.
  2. **Inorder**: traverse the left subtree, visit the node and then traverse its right subtree
  3. **Postorder**: first traverse the left subtree, traverse the right subtree, and then visit the node.

Assume pointer to root.

Need only simply linked lists,

Inorder-Tree-Walk (x)
IF $x \neq$ `Nil`
    Then Inorder-Tree-Walk($left(x)$)
        print($key(x)$)
        Inorder-Tree-Walk($right(x)$)

Preorder-Tree-Walk (x)
IF $x \neq$ `Nil`
    Then print($key(x)$)
        Preorder-Tree-Walk($left(x)$)
        Preorder-Tree-Walk($right(x)$)

Postorder-Tree-Walk (x)
IF $x \neq$ `Nil`
    Then Postorder-Tree-Walk($left(x)$)
        Postorder-Tree-Walk($right(x)$)
        print($key(x)$)

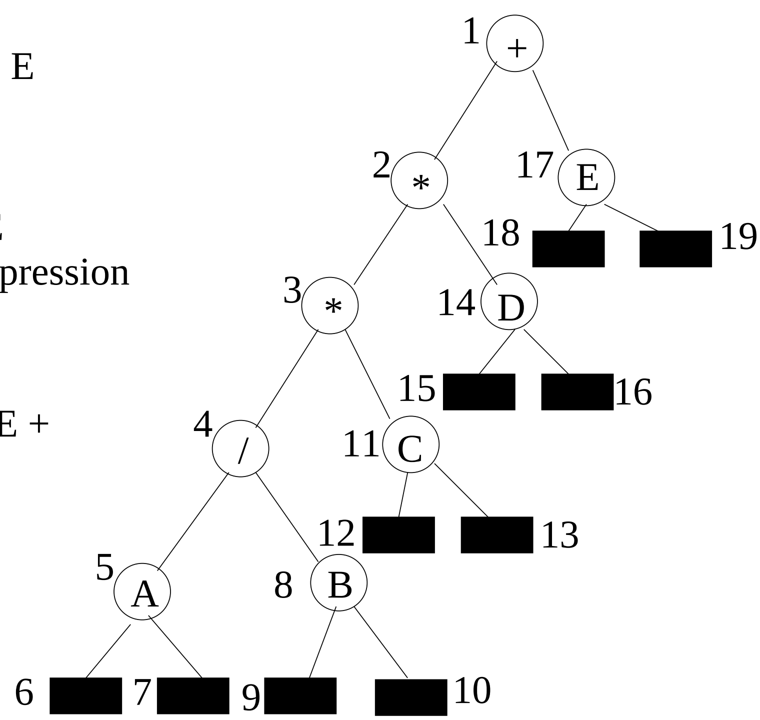
Binary-tree traversal: example

- **Preorder**: visit a node, then traverse its left subtree, and then traverse its right subtree.
- **Inorder**: traverse the left subtree, visit the node and then traverse its right subtree.
- **Postorder**: first traverse the left subtree, traverse the right subtree.

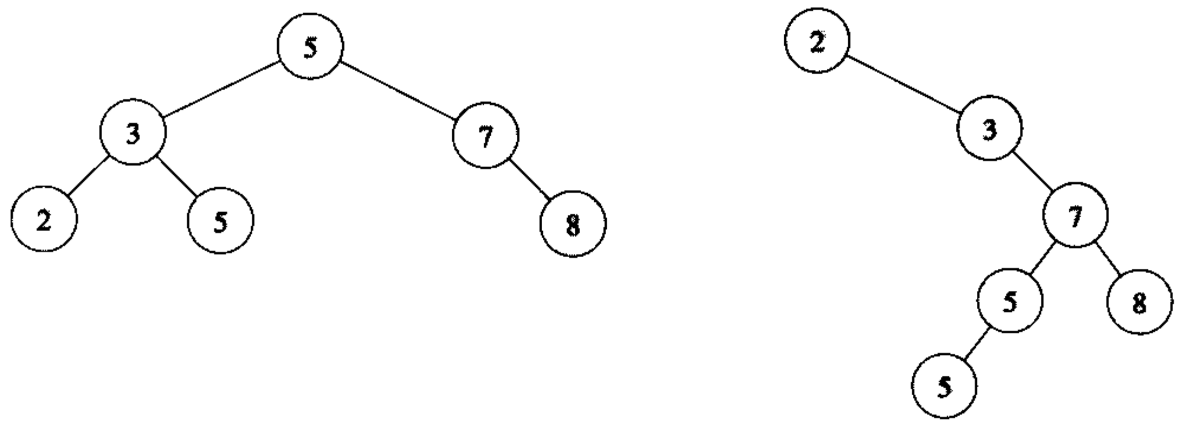
Preorder: + * * / A B C D E

Inorder:A / B * C * D + E
nfix form of the expression

Postorder: A B / C * D * E +



Binary-search-tree property



(a) (b)

Let x be a node in a binary search tree.
If y is a node in the left subtree of x, then $key[y] \leq key[x]$. If y is a node in the right subtree of x, then $key[x] \leq key[y]$.

Inorder traversal

simple recursive algorithm that prints out all the keys in a binary search tree in sorted order, thanks to **binary-search-tree property**

```
Inorder-Tree-Walk (x)
IF x ≠ Nil
   Then Inorder-Tree-Walk(left(x))
        print(key(x))
        Inorder-Tree-Walk(right(x))
```