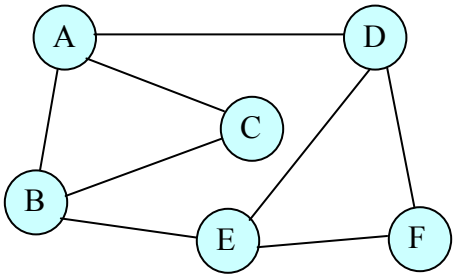
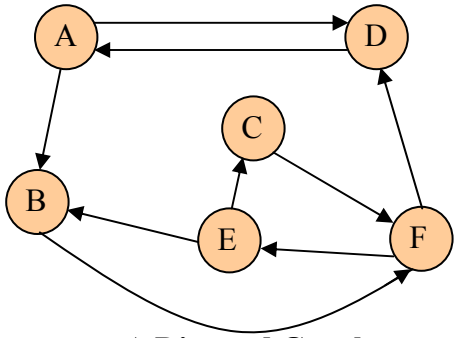
Graphs
A **graph** is a set of **vertices** and a set of **edges** connecting the vertices.
A **path** is a list of vertices connected by edges (that you can follow).

Examples



An Undirected Graph



A Directed Graph

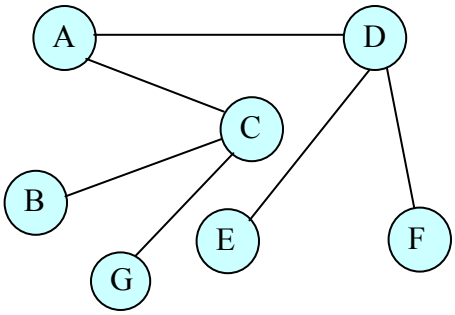
Paths

ABCAD
FEBCAD
CACAC

ADA
ABFECFD
ECFECF

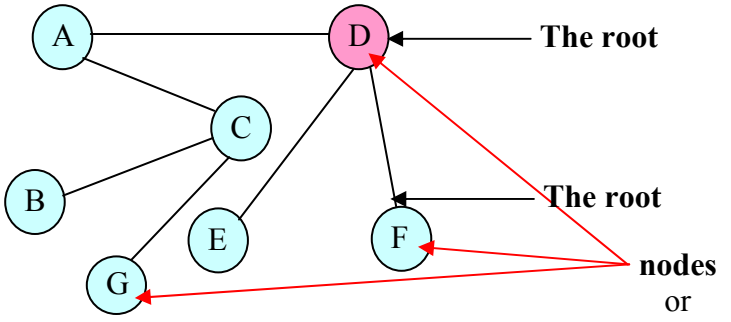
Trees
A **tree** is a *connected* graph without any *loops* (closed paths).

Examples
Neither of the graphs above is a tree.



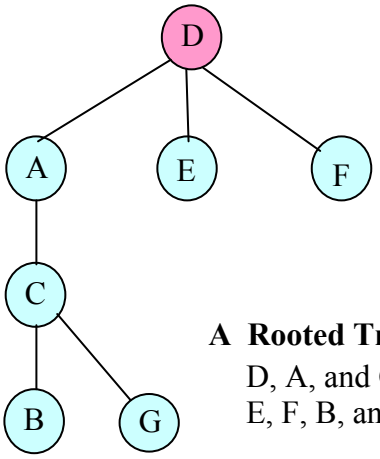
An Unrooted Tree

The tree to the left is an *unrooted* tree. Usually, when we work with trees, there is a special node designated as the **root** of the tree.



A Rooted Tree

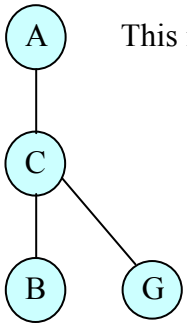
A Tree Hanging from its Root



A Rooted Tree Hanging from its Root
D, A, and C are **internal nodes**; they have children.
E, F, B, and G are **leaves**; they have no children.

The **children** of **D** are **A**, **E**, and, **F**. **A**, **E**, and, **F** are siblings. The **parent** of **B** is **C**. **B**, **G**, **E**, and **F** are leaves of the tree. They have no children. The **height** of the tree above is 3.

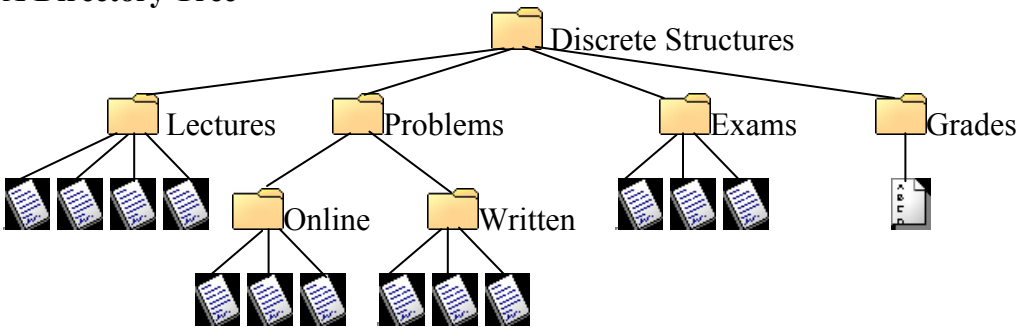
If the order of the children matters, the tree is called an **ordered tree**.



This is a **subtree** of the tree above. It has height 2.

This is a tree of height 0.

A Directory Tree



Theorem: A tree with n nodes has n - 1 edges.
Proof by Induction:
Base Case: A tree with 1 node has 0 edges.
Induction step**:**
Induction Assumption: Every tree with *k* nodes has *k - 1* edges.
Suppose a tree T has *k + 1* nodes. T must have a leaf L. Let T' be the tree obtained by removing L from T along with the edge that connected L to its parent. T' has *k* node. So by the induction assumption, T' has *k - 1* edges. T has just one more edge than T' so T has *k* edges. Q.E.D.

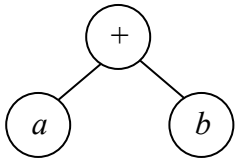
Binary Trees
A **binary tree** is either an empty tree or an ordered tree in which every node has exactly two subtrees, called the **left** and **right subtrees**.

Expression Trees
An algebraic expression can be represented by a binary tree.

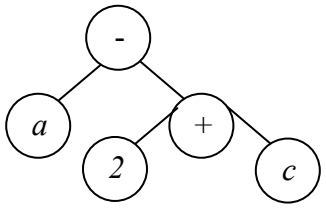
$a+b$ corresponds to the tree

```
   +
  / \
 a   b
```

or



$a-(2+c)$ corresponds to the tree

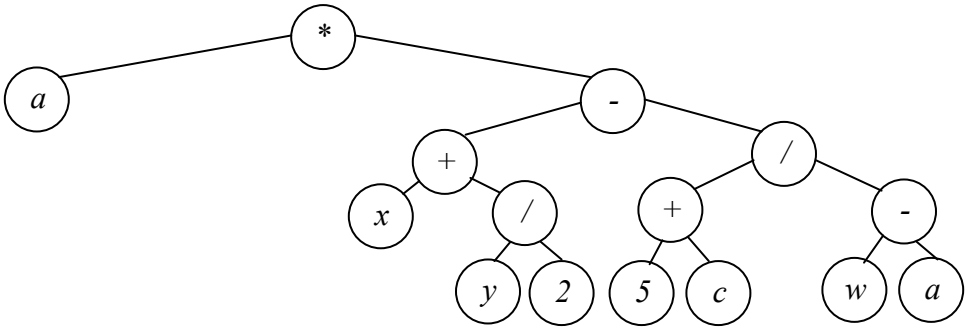


Notice that there are operators, e.g. +, -, *, or /, at all the internal nodes and literals, or numbers are at the leaves. In an expression tree, the order of the children is important (3 - 5 ≠5 - 3).

Example

Given the expression $a*\left(\left(x+y/2\right)-\dfrac{(5+c)}{(w-a)}\right)$,

a) Give the corresponding expression tree.



b) Give the Scheme expression that corresponds to the tree.

(* a (- (+ x (/ y 2)) (/ (+ 5 c) (- (w a))))

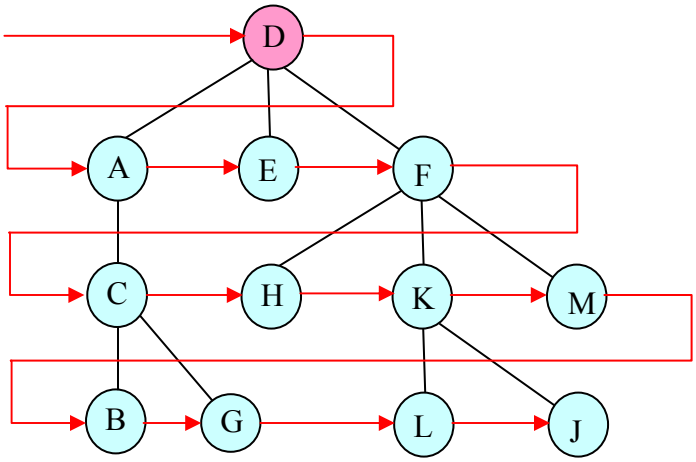
Notice that the Scheme expression, E(T) of an expression tree T is given by:
(root-operation E(left subtree of T) E(right subtree of T))

Traversing Binary Trees

To *traverse* a tree (or graph), you must visit every node of the tree (or graph). What you or your program does when it visits a node will depend on the application. If the nodes contain student records, a visit might entail entering the current semester's final grades or printing a tuition bill. When discuss traversal methods, we usually indicate a visit by writing the name of the node.

There are four common methods of traversing an ordered rooted tree.

1. Level Order Traversal

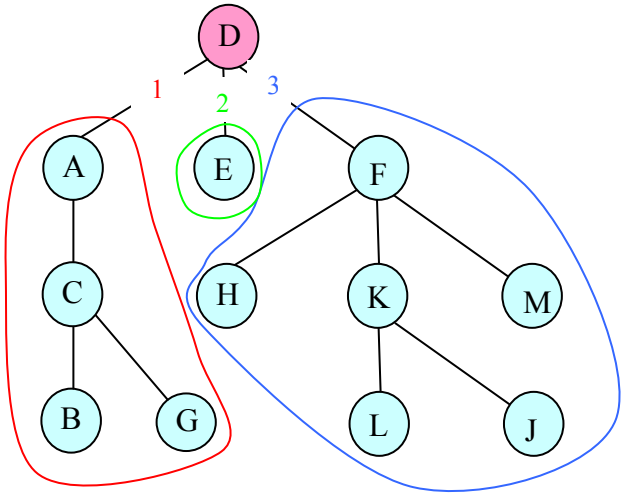


To traverse a tree in *level order*, visit the root (level 0) and then visit the nodes on each successive level from left to right. If we traverse the tree above in level order, we will visit the nodes as follows:

D A E F C H K M B G L J

2. Preorder Traversal

visit node D
then
preorder Traverse
subtree 1
preorder Traverse
subtree 2
preorder Traverse
subtree 3



To traverse a tree in *preorder*, visit the root, then do a preorder traversals of subtrees of the root from left to right. If we do a preorder traversal of the tree above, we will visit the nodes as follows:

D A C B G E F H K L J M

1 2 3

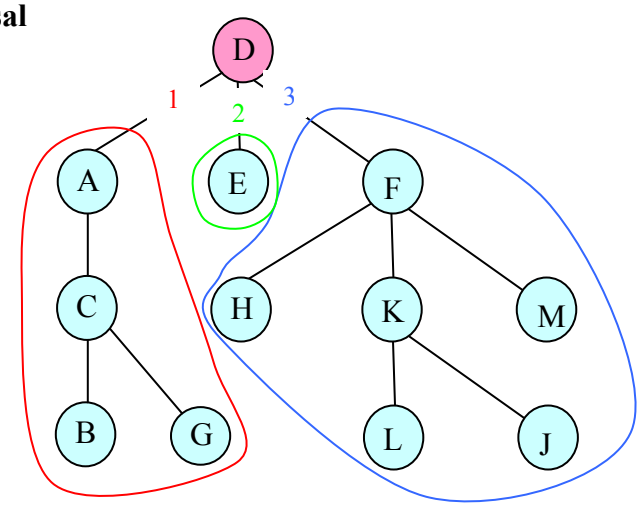
If T is a binary tree, `preorder(T)` is given by

```
preorder(T) {
  if (T is not null) {
     visit(root(T));
     preorder(left(T));
     preorder(right(T));
  }
}
```

If we traverse a binary expression tree in preorder we get a scheme-like representation of the expression with no parentheses. . The parentheses are not really needed to parse the expression and calculate its value.

3. Inorder Traversal

inorder Traverse
  subtree 1
then
visit node D
then
inorder Traverse
  subtree 2
inorder Traverse
  subtree 3



To traverse a tree *inorder*, do an inorder traversal of the leftmost subtree, then visit the root, then do inorder traversals of remaining subtrees of the root from left to right. If we do an inorder traversal of the tree above, we will visit the nodes as follows:

B C G A **D** E H F L K J M

1 2 3

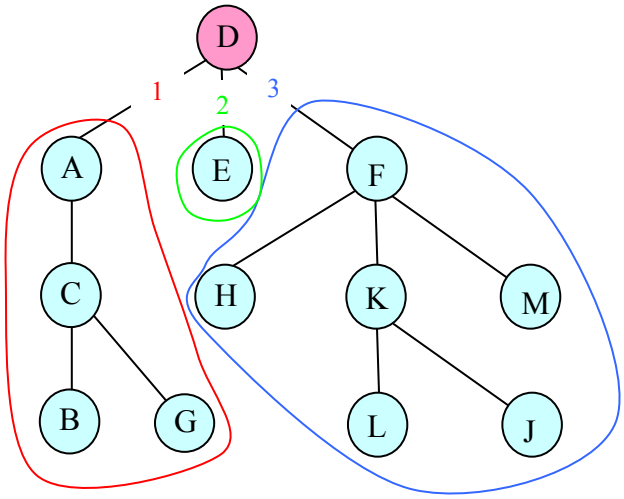
If T is a binary tree, `inorder(T)` is given by

```
inorder(T) {
  if (T is not null) {
     inorder(left(T));
     visit(root(T));
     inorder(right(T));
  }
}
```

If we traverse a binary expression tree inorder we get an expression in "infix" notation but parentheses may be necessary to make the expression really correspond to the tree.

4. Postorder Traversal

postorder Traverse
  subtree 1
postorder Traverse
  subtree 2
postorder Traverse
  subtree 3
then
visit node D



To traverse a tree in *postorder*, do postorder traversals of subtrees of the root from left to right., then visit the root. If we do a postorder traversal of the tree above, we will visit the nodes as follows:

B G C A E H L J K M F **D**
1 2 3

If T is a binary tree, `postorder(T)` is given by

```
postorder(T) {
  if (T is not null) {
     postorder(left(T));
     postorder(right(T));
     visit(root(T));
  }
}
```

If we traverse a binary expression tree postorder we get an expression in reverse Polish notation. This order is used for input on HP calculators. No parentheses are necessary to parse the expression and calculate its value.