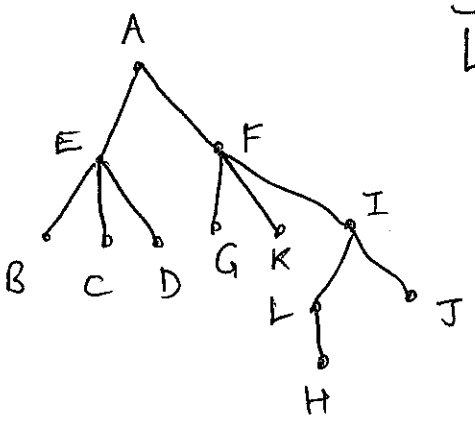
Trees – acyclic, connected graph – composed of nodes & edges.
(i) Unrooted (or Unoriented) tree (ii) Rooted tree.
We will discuss unrooted trees later.

Rooted tree: Recursively defined as root node attached to zero or more subtrees. Nodes with no children are called "leaves". Other nodes are internal nodes.

Example:



Root = A

Leaves = {B, C, D, G, K, H, J}

Children of A = {E, F} – E and F are siblings

Parent of C = E

Descendants of F = {F, G, K, I, H, J, L}

Ancestors of I = {I, F, A}

Proper descendants of node = All descendants of the node except itself

Similarly proper ancestors are all ancestors except itself.

Encoding tree using parent array representation ("Up tree"):
For each node, store its parent.

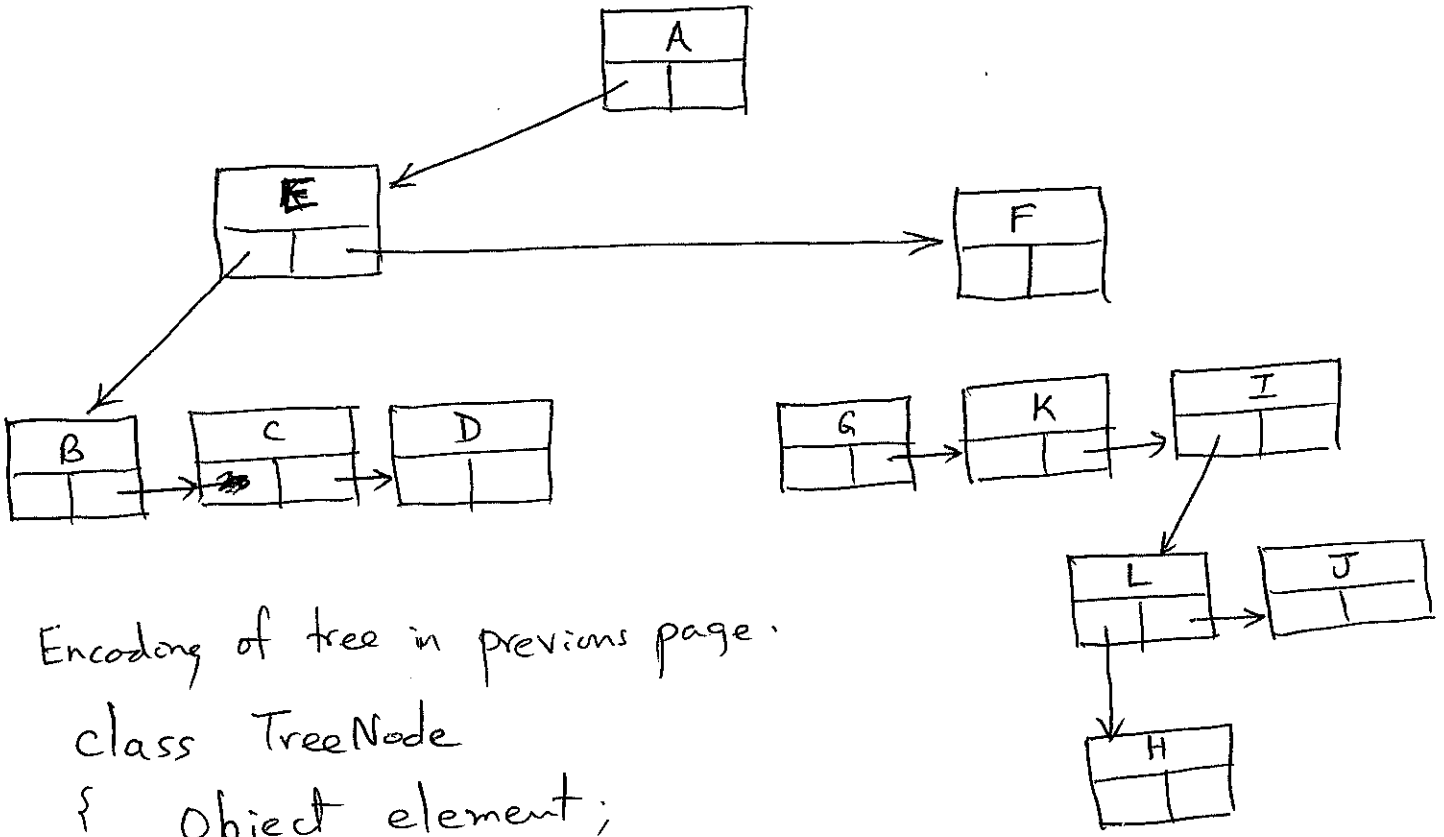
Ex:

A	B	C	D	E	F	G	H	I	J	K	L
–	E	E	E	A	A	F	L	F	I	F	I

← Parent array of above tree

This encoding is rarely used.

More common is to orient the edges of a tree downward, from parents to children. Each node has data and pointers (references) to left most child and right sibling.



Encoding of tree in previous page.

```
class TreeNode
{  Object element;
   TreeNode firstChild, nextSibling;
}
```

Path: A sequence of edges to travel from a node to another.
Example: F → I → L → H is a path of length 3 from F to H.

Length of a path is the number of its edges.

Depth of a node = Length of the path from the root to that node.

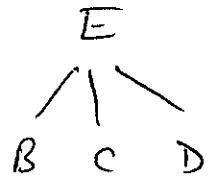
So, Depth (A) = 0, Depth (F) = 1, Depth (L) = 3 in above example.

Height of a node = Maximum length of a path from the node to any of its descendants, i.e; length of path from it to a farthest leaf that is a descendant of that node.

Height (G) = 0
Height (I) = 2

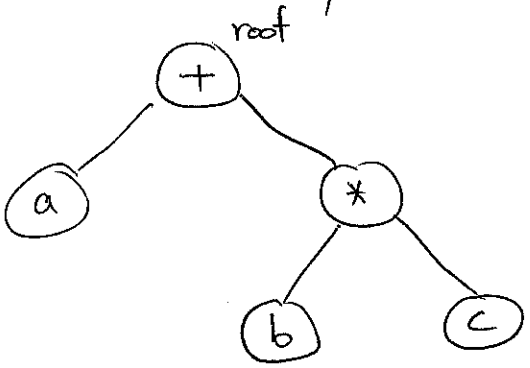
Subtree rooted at a node X is the part of the tree induced by X and its descendants.

Ex: Subtree ~~inst~~ at node E:



Binary Trees: Special subclass of trees in which each node has at most 2 children.

Ex: Arithmetic expression trees:




```
class BinaryTree
{ Object element;
  BinaryTree left, right;
}
```

Traversals: Visiting all nodes of a tree in some order.

```
Preorder (r)
// Visit a node before
// any of its proper desc
   if (r != null)
   {
        Visit r
        Preorder (r.left)
        Preorder (r.right)
   }
```

Output for example above:

+ a * b c

```
Postorder (r)
// visit a node after
// all of its proper descendants
   if (r != null)
   {   Postorder (r.left)
       Postorder (r.right)
       Visit r
   }
```

a b c * +

```
Inorder (r)
// Visit a node after all
// its proper descendants in
// left subtree but before
// its proper desc in right subtree
   if (r != null)
   {  Inorder (r.left)
      Visit r
      Inorder (r.right)
   }
```

a + b * c