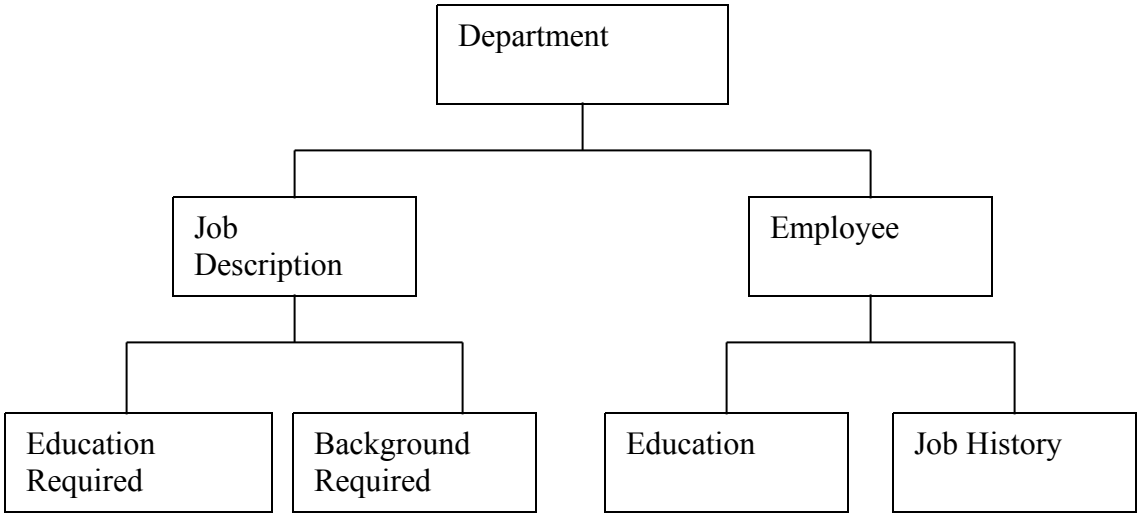
TREES

In Computer science the most important non linear structure is the tree. Although computer Scientists use trees as data/file structures, their properties were originally investigated by mathematicians and in fact, a substantial part of Graph Theory is devoted to the study of trees.

The use of tree structures in Computer Science dates back to the early 1950's when it was realized that the binary search could readily be represented by a binary tree.

The tree structure has hierarchical (top to bottom) and branching relationships between its nodes just like in the natural tree. Recall that nodes are "points" in memory containing data. An example of a tree structure is illustrated below:



Sometimes the need arises to store data in tree form within the computer – we will therefore examine their internal representation as well as suitable operations that can be performed on them.

Exercise

Study a typical COBOL record definition and represent it as a tree structure. Compare the COBOL representation to the FORTRAN or BASIC case.

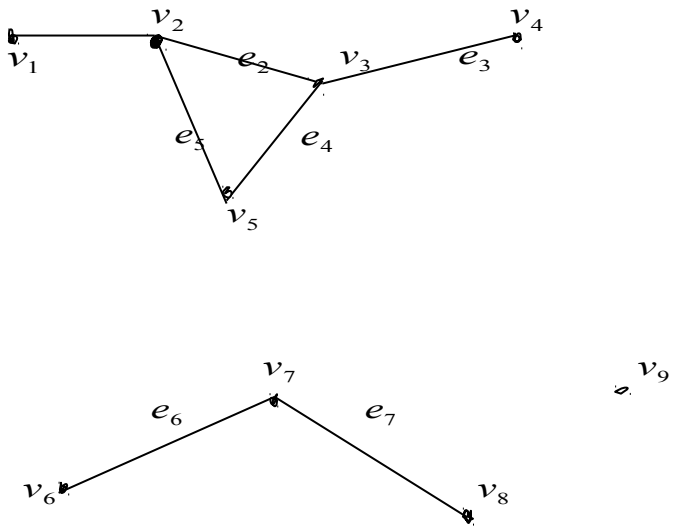
Trees, as data structures can be found in compiler designs, operating systems programs, file design programs such as ISAM, VSAM, database management systems and various application programs. They are extremely useful whenever data naturally fit together in a hierarchical structure. Even the arrangement of modules in a structured, top-down designed program forms a tree structure.

Preliminaries / Definitions

Graph

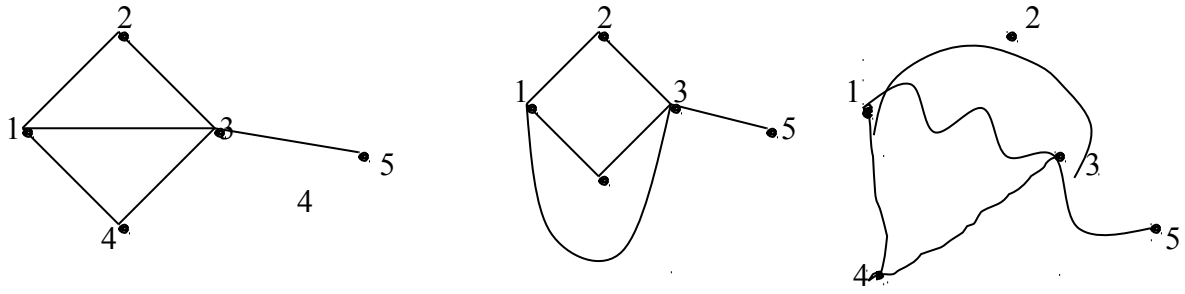
A graph G = (V,E) is a finite non-empty set V of vertices (points or nodes) and a set E of edges (lines, arcs or relationships) such that each edge e_k ϵ E is identified by an unordered pair (v_i, v_j) of vertices, where v_i ϵ V ; when i = j we call the edge a loop.

Shown below is a graph with vertices $v_1, v_2, \ldots, v_9$ and edges $e_1, e_2, \ldots, e_7$.



Pictorial Representation

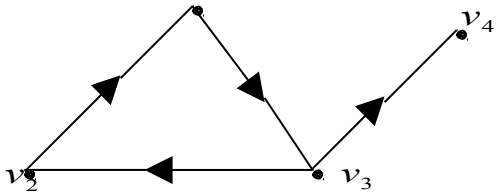
Any edge between vertices can be drawn as any kind of a curve (not necessarily a straight line). The important thing is the incidence between the edges and vertices; any intersection of edges is not a vertex unless specified. The same graph shown pictorially in three different ways is illustrated below.



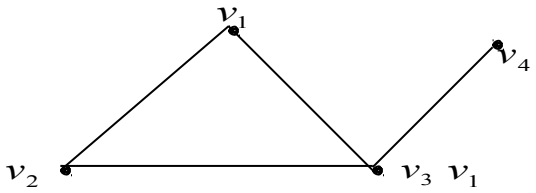
Directed and undirected graph

In a graph if the edges are identified with an ordered pair of vertices (v_1, v_j), then the graph is known as a directed graph or digraph; otherwise it is an undirected graph

v_1



(a) Directed graph



(b) Undirected graph

Degree of vertex

In an undirected graph, the **degree** of a vertex is the number of edges incident with the vertex. In a directed graph, if (v_1, v_j) is an edge, then v_1 is called the initial vertex and v_j is the final vertex. For a vertex, the **outdegree** is the number of out going edges for which the vertex is initial, and **indegree** is the number of incoming edges for which the vertex is final. These are illustrated for the above graphs

(a) Indegree of vertex v_1 = 1
Outdegree of vertex v_1 = 1
Indegree of vertex v_3 = 1
Outdegree of vertex v_3 = 2

(b) Degree of vertex v_3 = 3
Degree of vertex v_4 = 1
Degree of vertex v_1 = 2
Degree of vertex v_2 = 2

Null graph

A null graph is a graph consisting of isolated vertices, an isolated vertex being a vertex having no edges incident to it.

Path

A path is a sequence of nodes and edges $v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k$ in which each edge e_i connects the nodes v_{i-1} and v_i $(1 \leq I \leq k)$.
The implication of this definition is that the vertices are neighbour-wise adjacent and every vertex appears only once.

Path length

The path from v_i to v_k is said to be of length k-1. A special case is a loop, which is a path from itself to itself, and has a length of 0.

Simple path

A path is simple if all edges and all vertices on the path, except possibly the first and last vertices, are distinct.

Cycle or circuit

A cycle is a path in which the initial and final nodes coincide.

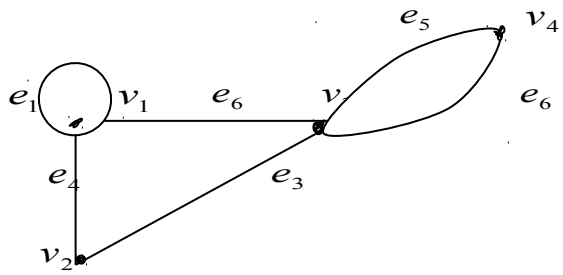
Connected graph

A connected graph G = (V,E) is said to be **connected** if there exists a path between any two vertices $v_i, v_j \in V$ regardless of how they have been selected, otherwise the graph is called disconnected.

Subgraph

A subgraph H = (V_1 , E_1) of G = (V,E) is a graph with $V_1 \subseteq V$ and $E_1 \subseteq E$, and each edge H has the same end vertices in H as in G.

In the graph below

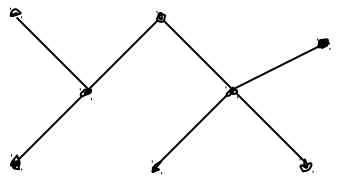


$v_1, e_2, v_3, e_3, v_2, e_4, v_1$ is a **cycle** or **circuit**. The edges e_5 and e_6 of the cycle v_3, e_6, v_4, e_5, v_3 are called multiple. v_1, e_1, v_1 is a loop.

Numerous definitions of trees exist. We now give one through the graph concept.

Tree

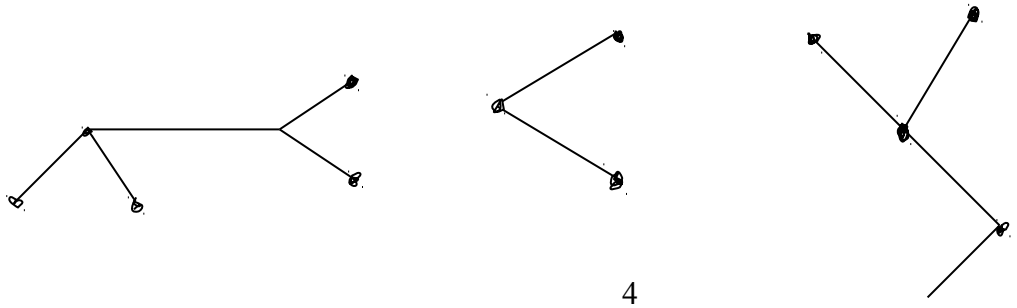
A tree is a connected graph with no cycles (and has at most one branch entering each node).



A tree

Forest

A collection of disjoint trees is called a forest.



A forest

Basic properties of trees

From the above definition, we have the following properties

(i) The path between any two nodes is unique and the length of a path is also unique. Further, if two paths have the same final node then one is a subgraph of the other.

(ii) A tree with n nodes contains n-1 lines. Conversely, any graph with n nodes and n-1 line is a tree.

Rooted trees

Since a tree contains no cycles, the length of a path in it is bounded. Therefore there exist **maximal paths** which are **proper** subpaths of no longer paths. The initial and final nodes of a maximal path are called the **root** and the **leaf** (terminal node) of the tree respectively. A tree is said to be rooted if it has a particular node specially designated as the root. Because a maximal path in a tree starts from the root and ends at a leaf, it is convenient to think of a rooted tree as a **directed** graph

Level

The root is said to lie on the first level of the tree. The level of any other node is the number of nodes on the path from the root to that node. In general, a node which lies on the jth level, is said to lie at the end of a path (from the root) with length j-1.

Branch node

It is any node which is not a terminal node.

Subtree

Any node defines a **subtree** of which it is the root, consisting of itself and all other nodes reachable from it.

Recursive definition of tree

In the recursive definition a tree is defined in terms of itself. The basis is the definition of a tree with n nodes in terms of trees with less than n nodes. The sense in the definition is due to the presence of at least one tree which is not defined in terms of itself (i.e. a tree with one node).

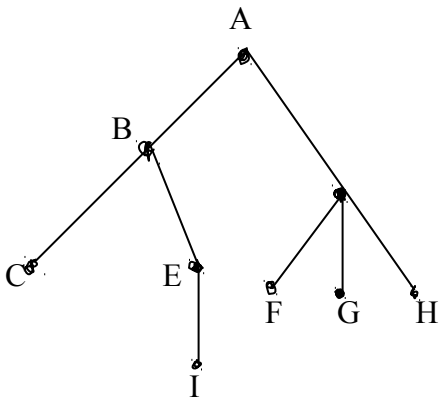
A tree is a finite set T of one or more node such that:

(i) there is a specifically designated node called the root of the tree, and

(ii) the remaining nodes are partitioned into m ≥ 0 disjoint sets $T_1, T_2, \ldots, T_m$ and each of these is In turn a tree. The trees T_i (i =1, 2, . . . , m) are called the subtrees of the root.

Useful terminology

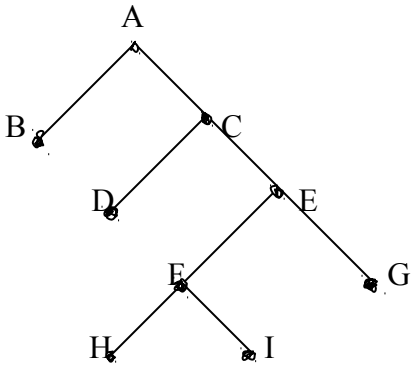
Names in family trees are often used to describe trees in Computer Science. In the father-son-brother terminology, if the root is the father, the nodes on level 2 are sons, those on level 3 are grandsons, etc. the nodes on all paths leading to another node A are the ancestors of A and the nodes on all paths leading from A are its descendents. in the following tree, we have the indicated relation of E.



I - son
D - brother
B - father
A - grandfather
C - 'uncle'
F,G,H - cousins
F,G,H - siblings of each other

Binary tree

A tree in which each node has exactly zero or two subtrees is a binary tree.



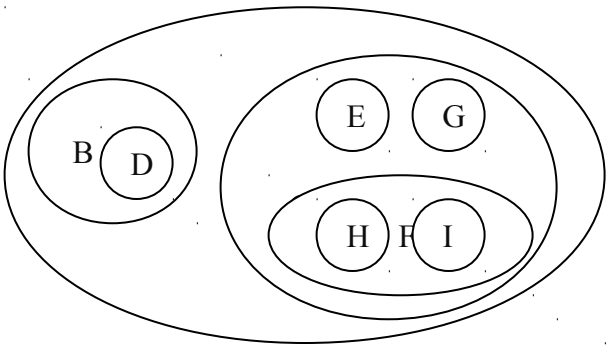
A binary tree

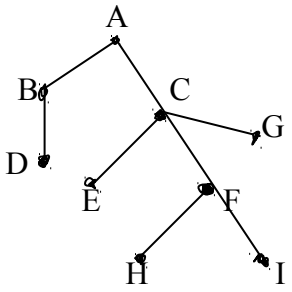
Note;
This definition for a binary tree is not universal. The other definition says that it is a tree in which each node has 0, 1 or 2 subtrees.
All algorithms we will look at will apply to both definitions.

Graphical representation of trees

1. **Nested representation**

(i) **Nested sets**





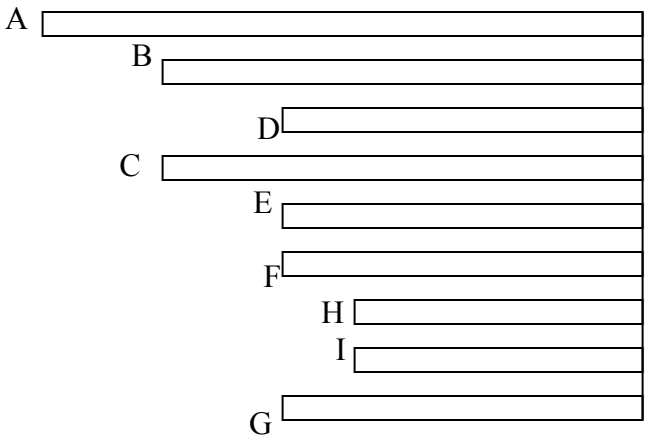
Each node is shown as a set, while its offspring are shown as disjoint subsets of parent node.

(ii) **Nested parenthesis**

(A (B (D)) (C (E) (F (H) (I)) (G)))

In this representation (X) is the tree with root X and $(X(X_1)(X_2),..., (X_n))$ is the tree with root X and has offspring $X_1, X_2,..., X_n$.

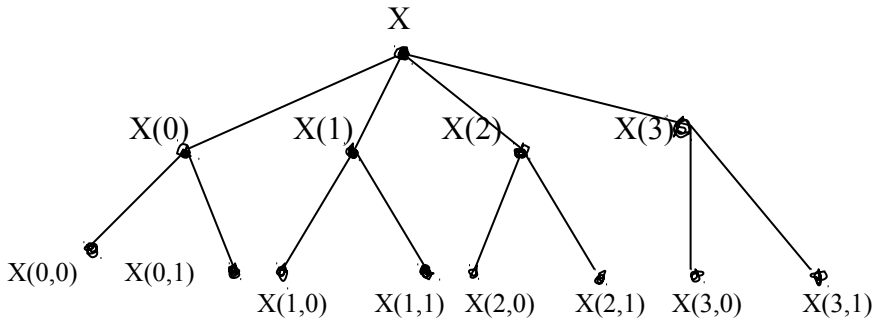
2. **Indentation**



Tree structures in practice

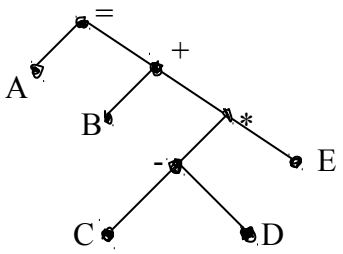
Any system with hierarchical relationships between its elements has a tree structure. The tree is a useful tool for analysing logical possibilities.

(1) **Arrays:** Arrays may be represented in tree form. The rectangular array gives the simplest form with three levels, the rows of the array corresponding to the nodes on the second level. The representation does not however display the column relationships between the elements in the matrix structure. The array X(I,J), $0 \leq I \leq 3$, $0 \leq J \leq 1$ could be represented as :



2. **Programs**

(i) **Arithmetic expression**: An arithmetic expression can be represented as a tree structure: the operators are branch nodes and the operands are leaves. For example, A = B + (C-D) * E can be represented as the tree:



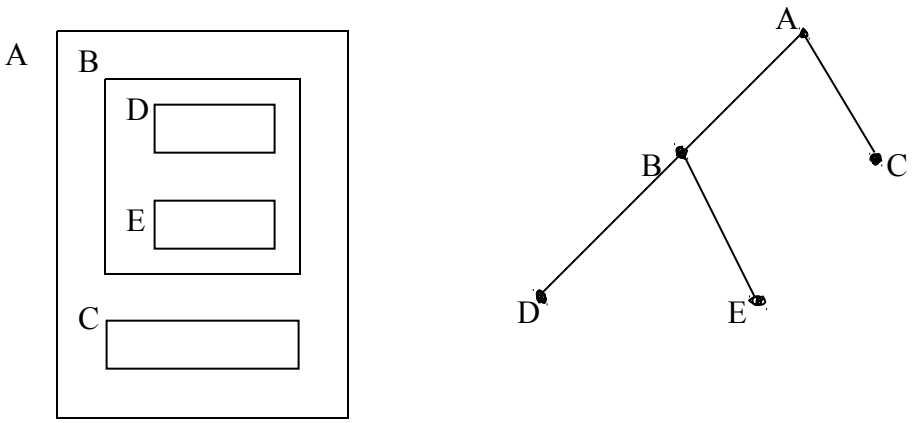
The order in which the operations will be performed is reflected in the tree diagram

(ii) **Compilers** - During translation of a source program into machine code, the former is put into an internal representation. This may be in tree form, representing the syntax of the source program. The tree is called a **parse** tree. This tree is then used to generate a sequence of machine language instructions.

(iii) **Operating systems** -Disk directories implemented by operating systems such as MS DOS is represented as a tree structure. Tree search is therefore used to locate information.

(iv) **Structured programs** - A structured program may be represented by a tree in which the nodes are the blocks of which the program is constructed, and a branch from node A to node B, is interpreted as meaning that B is a subblock of A. Processing the program can then be viewed as processing or traversing the tree in

some form.



3. **Decision and probability trees**

An important application of tree diagrams is in the logical analysis if decision making. Another is in probability, where the probability tree is useful for the evaluation of compound probabilities. E.g. zip code.

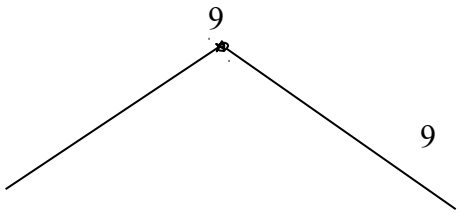
4. **Classification schemes**

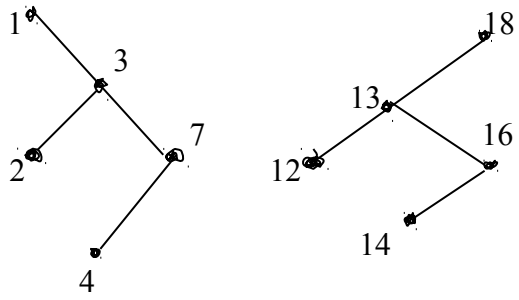
The system is considered as belonging to one large class; it is split into subclasses, and so on until levels are reached where subclasses are no longer split up. Examples are:

1. Library classification
2. Numbering of sections in books

5**. Searching**

The keys to be searched are represented in a tree called a binary search tree (BST) in which, with respect to the key of each node, the key of the left is numerically less, while that of the right is numerically greater. The keys 9,1,3,7,18,2,13,16,14,12,4 give the BST.





6. **Sorting** : in the list processing sort, the keys are first represented as a BSST. An infix traversal Through the BST then produces the keys in ascending sequence.

Tree traversal

Our discussion will be restricted to the binary tree since general trees can be represented by corresponding binary trees.

Traversing or walking through a tree is a method of systematically examining the nodes of the tree so that each node is visited only once. The effect of traversing a tree is a linear arrangement of the nodes according to the type of traversal used.

Method of traversing binary trees

These are defined recursively as;

Infix traversal

Traverse left subtree
Visit root
Traverse right subtree

Prefixed traversal

Visit root
Traverse left subtree
Traverse right subtree

Suffix traversal

Traverse left subtree
Traverse right subtree
Visit root

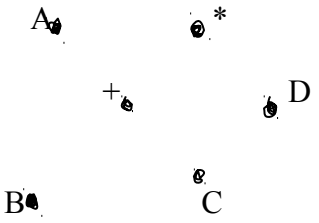
Consider the above algorithms applied to the binary trees below

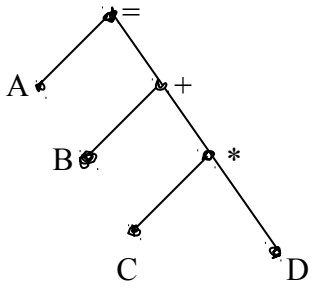
Case (a)

=

case (b)

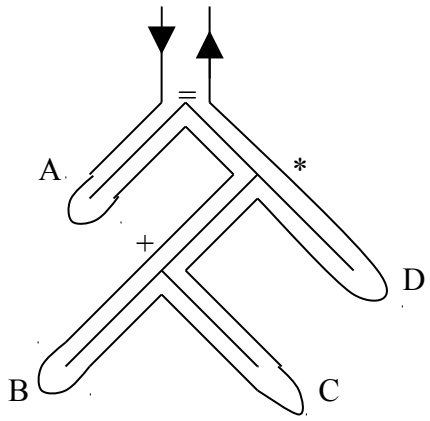
10

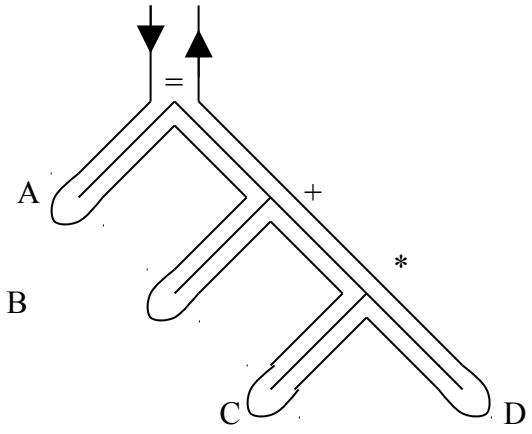




Infix:	A= B + C * D	A = B + C * D
Prefix:	=A * + BCD	= A + B * CD
Suffix:	ABC+ D * =	ABCD* + =

The above indicate that the infix traversal of the two binary tree with different structures lead to the same linear arrangement of the nodes. This creates ambiguity in the evaluation of an infix expression. The ambiguity in the infix traversal can be removed by the following scheme:
Consider the tree as a solid wall, walk round the tree keeping the wall to the left as indicated





Output terminal nodes as they are entered
Write a left parenthesis on meeting a non terminal node for the first time.
Output a non terminal node when it is encountered for the second time
Write a right parenthesis when a non terminal node is encountered for the third time.

Thus we get the following for the two cases
(A = ((B + C) * D)) (A = (B + (C * D)))

The above expressions are said to be fully parenthesized, i.e. each operator and its operands are enclosed in a pair of parentheses. They however have an over abundance of parentheses since the expressions could be evaluated correctly with fewer parentheses.

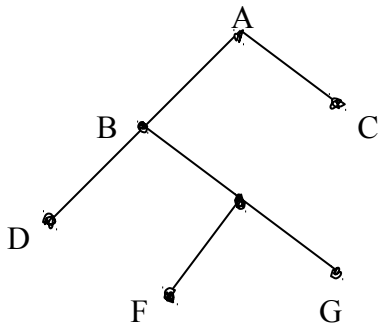
Redundant parentheses may be avoided by providing the left and right parentheses for an operator encountered only when it has a lower priority than its father. The actions in 1 and 3 remain unchanged. This modification leads to revised expressions:
A = (B + C) * D A = B + C * D
which correctly represent the values of the expressions represented by the trees.

Representation of the binary tree within memory

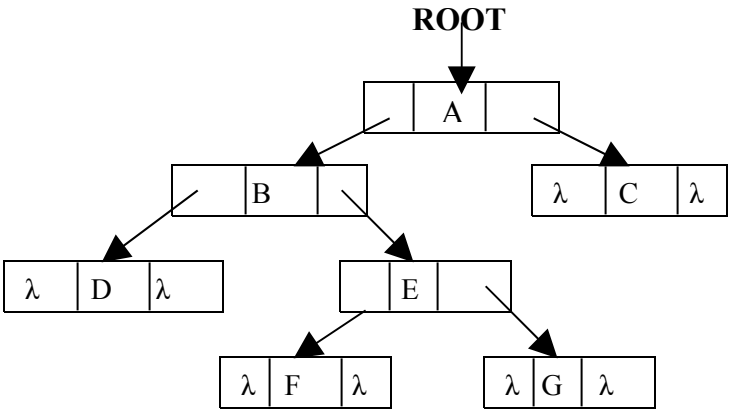
Each node is represented by an information field, a left link field and a right link field. The left link contains the address of the root of the left subtree of the node under consideration, while the right link contains the address of the root of the corresponding right subtree. The link fields of terminal nodes contain the null link λ. The variable ROOT called the pointer to the tree contains the address of the root of the tree. If ROOT = λ, then the tree is empty

Typical node

typical binary tree



Representation of the binary tree



Since the traversal algorithms are recursive, the stack is employed to store the root address of the subtree that currently has to be traversed.

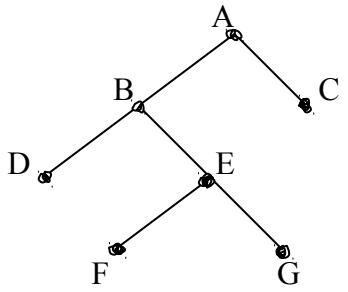
Infix traversal algorithm

STACK - stack
X - address (link) variable
ROOT - root pointer
T - stack pointer
LLINK - left link
RLINK - right link
λ - null link

S1. T ← 0

X ← ROOT
S2. If X = λ, go to S4 endif
S3 T ← T + 1
STACK(T) ← X
X ← LLINK(X)
Go to S2
S4. If T = 0, stop endif
S5. X ← STACK(T)
T ← T – 1
S6. Visit X
X ← RLINK(X)
Go to S2

Exercise
Execute this algorithm given the tree:



Exercise
Write a corresponding algorithm for the prefix traversal of a binary tree.

Suffix traversal algorithm

S1. T ← 0
X ← ROOT
S2. If X = λ, go to S4 endif
S3 T ← T + 1
STACK(T) ← X
X ← LLINK(X)
Go to S2
S4. If T = 0, stop endif

X ← STACK(T)
T ← T – 1
If X = +ve, go to S5 endif
X ← − X
Visit node X
Go to S4

S5. T ← T + 1
STACK(T) ← − X
X ← RLINK(X)
Go to S2