Module 8: Trees and Graphs

Theme 1: Basic Properties of Trees

A (rooted) **tree** is a finite set of nodes such that

- there is a specially designated node called the **root**.
- the remaining nodes are partitioned into d disjoint sets $T_1, T_2, \ldots, T_d$ such that each of these sets is a tree. The sets $T_1, T_2, \ldots, T_d$ are called **subtrees**, and d the degree of the root.

The above is an example of a recursive definition, as we have already seen in previous modules.

Example 1: In Figure 1 we show a tree rooted at A with three subtrees T_1, T_2 and T_3 rooted at B, C and D, respectively.

We now introduce some terminology for trees:

- A tree consists of **nodes** or **vertices** that store information and often are labeled by a number or a letter. In Figure 1 the nodes are labeled as $A, B, \ldots, M$.
- An **edge** is an unordered pair of nodes (usually denoted as a segment connecting two nodes). For example, (A, B) is an edge in Figure 1.
- The number of subtrees of a node is called its **degree**. For example, node A is of degree three, while node E is of degree two. The maximum degree of all nodes is called the degree of the tree.
- A **leaf** or a **terminal node** is a node of degree zero. Nodes K, L, F, G, M, I and J are leaves in Figure 1.
- A node that is not a leaf is called an **interior node** or an **internal node** (e.g., see nodes B and D).
- Roots of subtrees of a node X are called **children** of X while X is known as the **parent** of its children. For example, B, C and D are children of A, while A is the parent of B, C and D.
- Children of the same parent are called **siblings**. Thus B, C, D are siblings as well as K and L are siblings.
- The **ancestors** of a node are all the nodes along the path from the root to that node. For example, ancestors of M are H, D and A.
- The **descendants** of a node are all the nodes along the path from that node to a terminal node. Thus descendants of B are F, E, K and L.

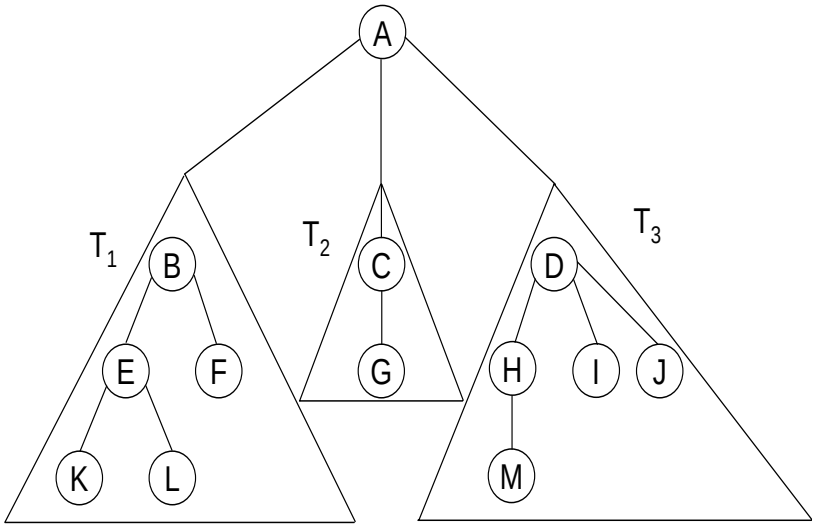


Figure 1: Example of a tree.

- The **level** of a node is defined by letting the root to be at level zero[1], while a node at level l has children at level $l + 1$. For example, the root A in Figure 1 is at level zero, nodes B, C, D are at level one, nodes E, F, G, H, I, J ate level two, and nodes K, L, M are at level three.

- The **depth** of a node is its level number. The **height** of a tree is the maximum level of any node in this tree. Node G is at depth two, while node M at depth three. The height of the tree presented in Figure 1 is three.

- A tree is called a d**-ary tree** if every internal node has no more than d children. A tree is called a **full** d**-ary tree** if every internal node has *exactly* d children. A **complete tree** is a full tree up the last but one level, that is, the last level of such a tree is not full. A **binary tree** is a tree with $d = 2$. The tree in Figure 1 is a 3-ary tree, which is neither a full tree nor a complete tree.

- An **ordered rooted tree** is a rooted tree where the children of each internal node are ordered. We usually order the subtrees from left to right. Therefore, for a binary (ordered) tree the subtrees are called the **left subtree** and the **right subtree**.

- A **forest** is a set of disjoint trees.

Now we study some basic properties of trees. We start with a simple one.

Theorem 1. *A tree with n nodes has $n - 1$ edges.*
Proof. Every node *except* the root has exactly one in-coming edge. Since there are $n - 1$ nodes other than the root, there are $n - 1$ edges in a tree.

[1]Some authors prefer to set the root to be on level one.

The next result summarizes our basic knowledge about the maximum number of nodes and the height.

Theorem 2. *Let us consider a binary tree.*

(i) *The maximum number of nodes at level* i *is* 2^i *for* $i \geq 0$.

(ii) *The maximum number of all nodes in a tree of height* h *is* $2^{h+1} - 1$.

(iii) *If a binary tree of height* h *has* n *nodes then*

$$h \geq \log_2(n+1) - 1.$$

(iv) *If a binary tree of height* h *has* l *leaves, then*

$$h \geq \log_2 l.$$

Proof. We first proof (i) by induction. It is easy to see that it is true for $i = 0$ since there is only one node (the root) at level zero. Let now, by the induction hypothesis, assume there are no more 2^i nodes at level i. We must prove that at level $i+1$ there are no more than 2^{i+1} nodes. Indeed, every node at level i may have no more than two children. Since there are 2^i nodes on level i, we must have no more than $2^i \cdot 2 = 2^{i+1}$ nodes at level $i+1$. By mathematical induction we prove (i).

To prove (ii) we use (i) and the summation formula for the geometric progression. Since every level has at most 2^i nodes and there are no more than h levels the total number of nodes cannot exceed

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1,$$

which proves (ii).

Now we prove (iii). If a tree has height h, then the maximum number of nodes by (ii) is $2^{h+1} - 1$ which must be at least be as big as n, that is,

$$2^{h+1} - 1 \geq n.$$

This implies that $h \geq \log_2(n+1) - 1$, and completes the proof of (iii). Part (iv) can be proved in exactly the same manner (see also Theorem 3 below).

Exercise 8A: Consider a d- ary tree (i.e., nodes have degree at most d). Show that at level k there are at most d^k nodes. Conclude the total number of nodes in a tree of height h is $(d^{h+1} - 1)/(d-1)$.

Finally, we prove one result concerning a relationship between the number of leaves and the number of nodes of higher degrees.

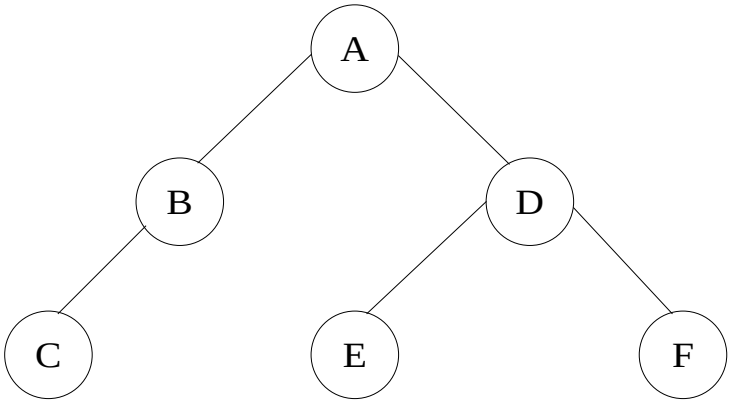


Figure 2: Illustration to Theorem 3.

Theorem 3. *Let us consider a nonempty binary tree with n_0 leaves and n_2 nodes of degree two. Then*

$$n_0 = n_2 + 1.$$

Proof. Let n be the total number of all nodes, and n_1 the number of nodes of degree one. Clearly

$$n = n_0 + n_1 + n_2.$$

On the other hand, if b is the number of edges, then — as already observed in Theorem 1 — we have $n = b + 1$. But also

$$n = b + 1 = n_1 + 2n_2 + 1$$

Comparing the last two displayed equations we prove out theorem.

In Figure 2 the reader can verify that $n_2 = 2$, $n_1 = 1$ and $n_0 = 3$, hence $n_0 = n_2 + 1$ as predicted by Theorem 3.

Theme 2: Tree Traversals

Trees are often used to store information. In order to retrieve such information we need a procedure to visit all nodes of a tree. We describe here three such procedures called *inorder*, *postorder* and *preorder* traversals. Throughout this section we assume that trees are ordered trees (from left to right).

Definition. Let T be an (ordered) rooted tree with $T_1, T_2, \ldots T_d$ subtrees of the root.

1. If T is null, then the empty list is preorder, inorder and postorder traversal of T.
2. If T consists of a single node, then that node is preorder, inorder and postorder traversal of T.
3. Otherwise, let $T_1, T_2, \ldots T_d$ be nonempty subtrees of the root.

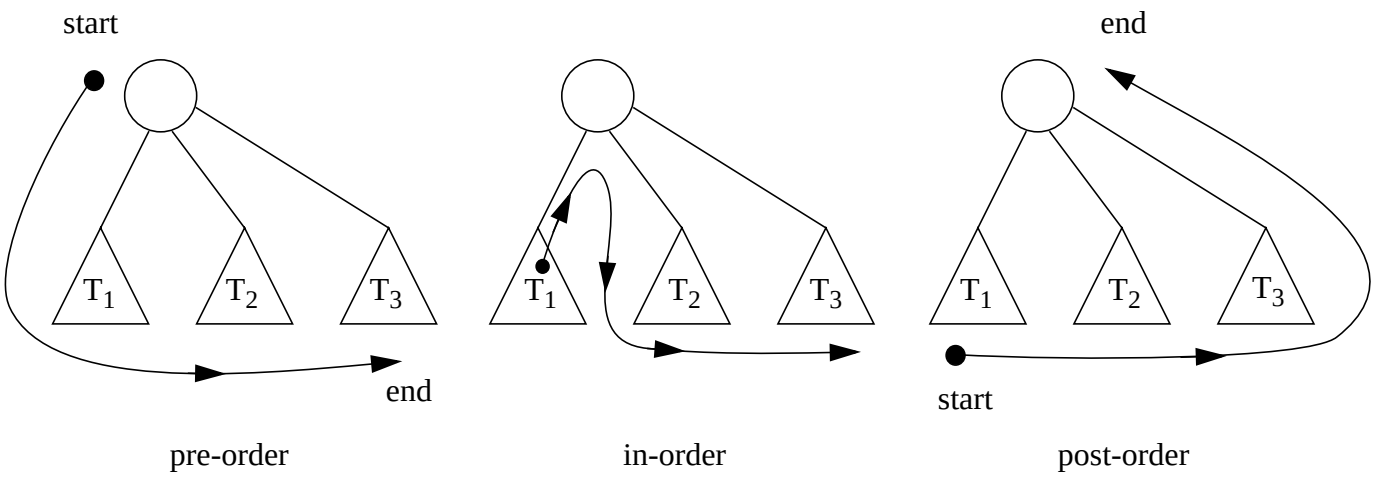


Figure 3: Illustration to preorder, inorder and postorder traversals.

- The **preorder traversal** of nodes in T is the following: the root of T followed by the nodes of T_1 in preorder, then nodes of T_2 in preorder traversal, ..., followed by T_d in preorder (cf. Figure 3).
- The **inorder traversal** of nodes in T is the following: nodes of T_1 in inorder, followed by the root of T, followed by the nodes of $T_2, T_3, \ldots, T_d$ in inorder (cf. Figure 3).
- The **postorder traversal** of nodes in T is the following: nodes of T_1 in postorder, followed by $T_2, \ldots, T_d$ in postorder, followed by the root (cf. Figure 3).

Example 2: Let us consider the tree T in Figure 1. The root A has three subtrees T_1 rooted at B, T_2 rooted at C, and subtree T_3 rooted at D. The *preorder* traversal is

$$\text{preorder of T} = \{A, B, E, K, L, F, C, G, D, H, M, I, J\}$$

since after the root A we visit first T_1 (so we list the root B and visit subtrees of T_1), then subtree T_2 rooted at C and its subtrees, and finally we visit the subtree T_3 rooted D and its subtrees.

The *inorder traversal* of T is

$$\text{inorder of T} = \{K, E, L, B, F, A, G, C, M, H, D, I, J\}$$

since we first must traverse inorder the subtree T_1 rooted at B. But inorder traversal of T_2 starts by traversing in inorder the subtree rooted at E, which in turn must start at K. Since K is a single node, we list it. Then we move backward and list the root, which is E, and move to the right subtree that turns out to be a single node L. Now, we can move up to B, that we list next, and finally node A. Then we continue in the same manner.

Finally, the postorder traversal of T is as follows:

$$\text{postorder of T} = \{K, L, E, F, B, G, C, M, H, I, J, D, A\}$$

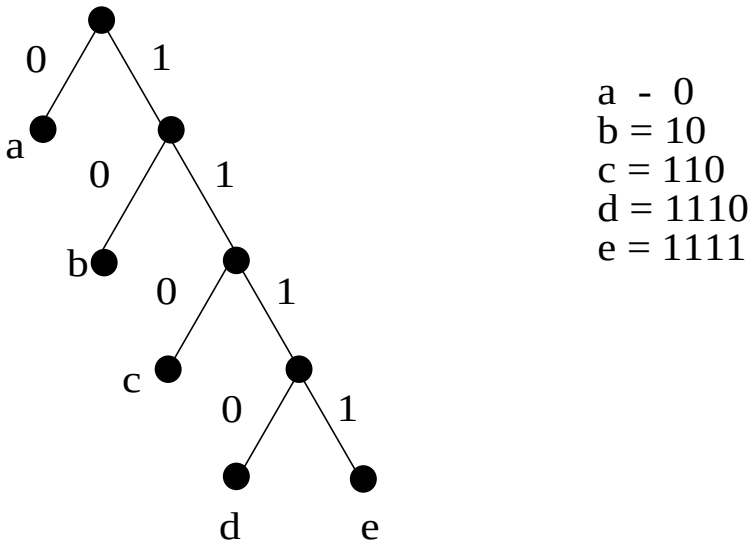


Figure 4: A tree representing a prefix code.

since we must first traverse postorder T_1, which means postorder traversal of a subtree rooted at E, which leads to E, L, and E. The rest follows the same pattern.

Theme 3: Applications of Trees

We discuss here two applications of trees, namely to build optimal prefix code (known as Huffman's code), and evaluations of arithmetic expressions.

Huffman Code

Coding is a mapping from a set of letters (symbols, characters) to a set of binary sequences. For example, we can set $A = 01,\ B = 0$ and $C = 10$ (however, as we shall see this is not a good code). But why to encode? The main reason is to find a (one-to-one) coding such that the length of the coded message is as short as possible (this is called *data compression*). However, not every coding is good, since – if we are not careful – we may encode a message that we won't be able to decode uniquely. For example, with the encoding as above, let us assume that we receive the following message

$$01001.$$

We can decode in many ways, for example as

$$BCA \quad \text{or} \quad ABA, \text{ etc.}$$

In order to avoid the above decoding problems, we need to construct special codes known as **prefix codes**.

> A code is called a **prefix code** if the bit string for a letter must never occur as the first part of the bit strings for another letter. In other words, no code is a prefix of another code. (By a prefix of the string $x_1x_2\ldots x_n$ we mean $x_1x_2\ldots x_i$ for some $1 \leq i \leq n$.)

It is easy to construct prefix codes using binary trees. Let us assume that we want to encode a subset of English characters. We build a binary tree with *leaves* labeled by these characters and we label the edges of the tree by bits 0 and 1, say, a left child of a node is labeled by 0 while a right child by 1. The code associated with a letter is a sequence of labels on the path from the root to the leaf containing this character. We *observe* that by assigning leaves to characters, we assure the prefix code property (if we label any internal node by a character, then the path or a code of this node will be a prefix of all other characters that are assigned to nodes below this internal node).

Example 3: In Figure 4 we draw a tree and the associated prefix code. In particular, we find that $a = 0$, $b = 10$, $c = 110$, $d = 1110$ and $e = 1111$. Indeed, no code is a prefix of another code. Therefore, a message like this

$$0111010101111$$

can be uniquely decoded as

$$adbbe.$$

It should be clear that there are many ways of encoding a message. But intuitively, one should assign shorter code to more frequent symbols in order to get on average as short code as possible. We illustrate this in the following example.

Example 4: Let $s = \{a, b, c, d, e\}$ be the set of symbols that we want to encode. The probabilities of these symbols and two different codes are shown in Table 1. Observe that both codes are prefix codes. Let us now compute the average code lengths L_1 and L_2 for both codes. We have

$$\begin{aligned} L_1 &= P(a)\cdot 3 + P(b)\cdot 3 + P(c)\cdot 3 + P(d)\cdot 3 + P(e)\cdot 3 = 3, \\ L_2 &= P(a)\cdot 3 + P(b)\cdot 2 + P(c)\cdot 2 + P(d)\cdot 3 + P(e)\cdot 2 = 2.2. \end{aligned}$$

Thus the average length of the second code is shorter, and – if there is no other constraint – this code should be used.

Let us now consider a general case. Let f_i, $1 \leq i \leq n$ be symbols with the corresponding probabilities $P(f_i)$. For a code C the average code length is defined as

$$L(C) = \sum_{i=1}^{n} P(f_i)|C(f_i)|$$

Table 1: Two prefix codes.

Symbol	Probability	Code 1	Code 2
a	0.12	000	000
b	0.40	001	11
c	0.15	010	01
d	0.08	011	001
e	0.25	100	10

where $|C(f_i)|$ is the length of the code assigned to f_i. Indeed, as discussed in Module 7, to compute the average of the code C we must compute the sum of products "frequency $\times$ length". We want to find a code C such that the average length $L(C)$ is as short as possible, that is,

$$\min_C \{L(C)\}.$$

The above is an example of a simple *optimization problem*: we are looking for a code (mapping from a set of messages S to a sequence of binary strings) such that the average code length $L(C)$ is the smallest. It turns out that this problem is easy to solve.

In 1952 Huffman proposed the following solution:

1. Select two symbols f_i and f_j that have the lowest probabilities, and replace them by a single (imaginary) symbol, say f_{ij}, whose probability is the sum of $P(f_i)$ and $P(f_j)$.

2. Apply Step 1 recursively until you exhaust all symbols (and the final total probability of the imaginary symbol is equal to one).

3. The code for the original symbols is obtained by using the code for f_{ij} (defined in Step 1) with 0 appended for the code for f_i and 1 appended for the code for f_j.

This procedure, which can be proved to be optimal, and it is best implemented on trees, as explained in the following example.

Example 5: We find the best code for symbols $S = \{a, b, c, d, e\}$ with probabilities defined in Table 1 of the previous example. The construction is shown in Figure 5. We first observe that symbols a and d have the smallest probabilities. So we join them building a small tree with a new node f_{da} of the total probability $0.12 + 0.08 = 0.2$. Now we have new set $S_1 = \{f_{da}, b, c, e\}$ with the probabilities $0.2, 0.4, 0.15, 0.25$, respectively. We apply the same algorithm as before. We choose two symbols with the smallest probabilities (a tie is broken arbitrarily). In our case it happens to be f_{da} and c. We

build a new node f_{dac} of probability 0.35 and construct a tree as shown. Continuing this way we end up with the tree shown in the figure. Now we read:

$$\begin{aligned} b &= 0 \\ e &= 10 \\ c &= 110 \\ d &= 1110 \\ a &= 1111. \end{aligned}$$

This is our Huffman code with the average code length

$$L = 0.4 + 2 \cdot 0.25 + 3 \cdot 0.15 + 4 \cdot 0.12 + 4 \cdot 0.08 = 2.15.$$

Observe that this code is better than the other two codes discussed in the previous example.

Evaluation of Arithmetic Expressions

Computers often must evaluate arithmetic expressions like

$$(A + B) * (C - D/F) \tag{1}$$

where A, B, C, D and F are called **operands** and $+, -, *$ and $/$ are called the **operators**. How to evaluate efficiently such expressions? It turns out that a tree representation may help transforming such arithmetic expressions into others that are easier to evaluate by computers.

Let us start with a computer representation. We restrict our discussion to *binary operators* (i.e., such that need two operands, like $A * B$). Then we build a binary trees such that:

1. Every leaf is labeled by an operand.
2. Every interior node is labeled by an operator. Suppose a node is labeled by a binary operand Θ (where $\Theta = \{+, -, /, *\}$) and the left child of this node represents expression E_1, while the right child expression E_2. Then the node labeled by Θ represents expression $(E_1)\ \Theta\ (E_2)$. The tree representing $(A + B) * (C - D/F)$ is shown in Figure 6.

Let us have a closer look at the **expression tree** shown in Figure 6. Suppose someone gives to you such a tree. Can you quickly find the arithmetic expression? Indeed, you can! Let us traverse *inorder* the tree in this figure. We obtain:

$$(A + B) * (C - (D/F)),$$

thus we recover the original expression. The problem with this approach is that we need to keep parenthesis around each internal expression. In order to avoid them, we change the **infix notation** to

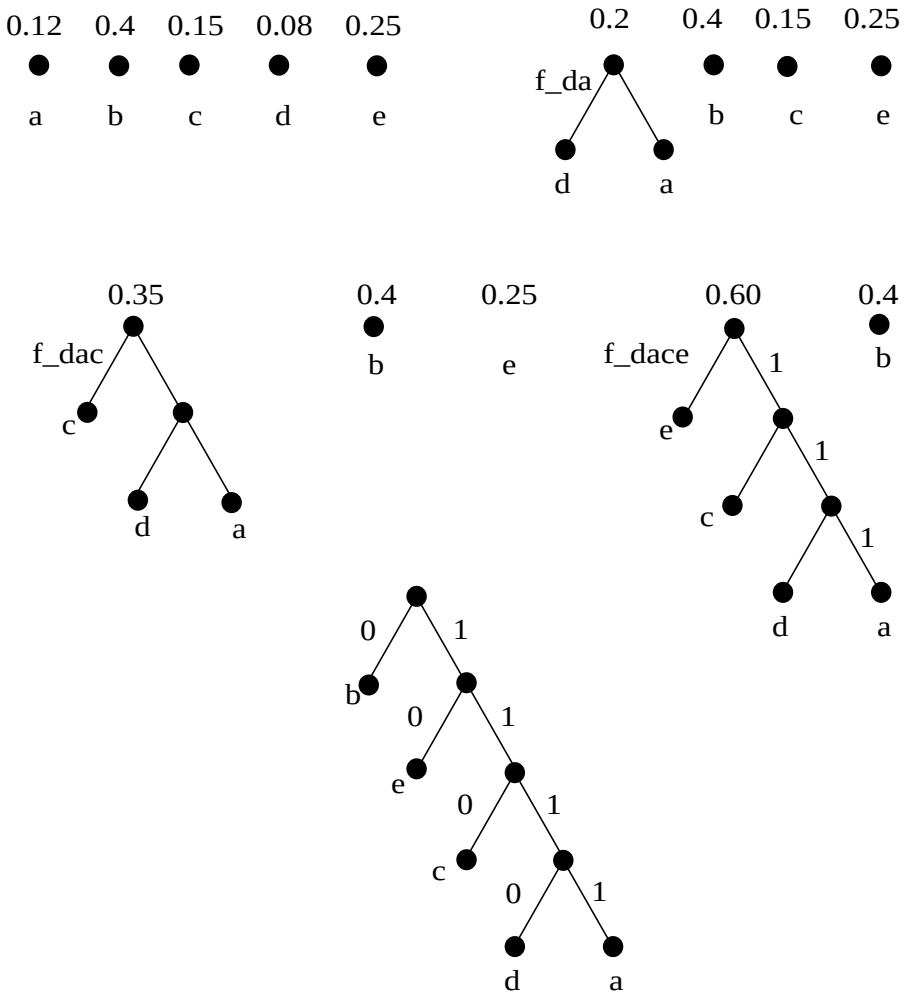


Figure 5: The construction of a Huffman tree and a Huffman code.

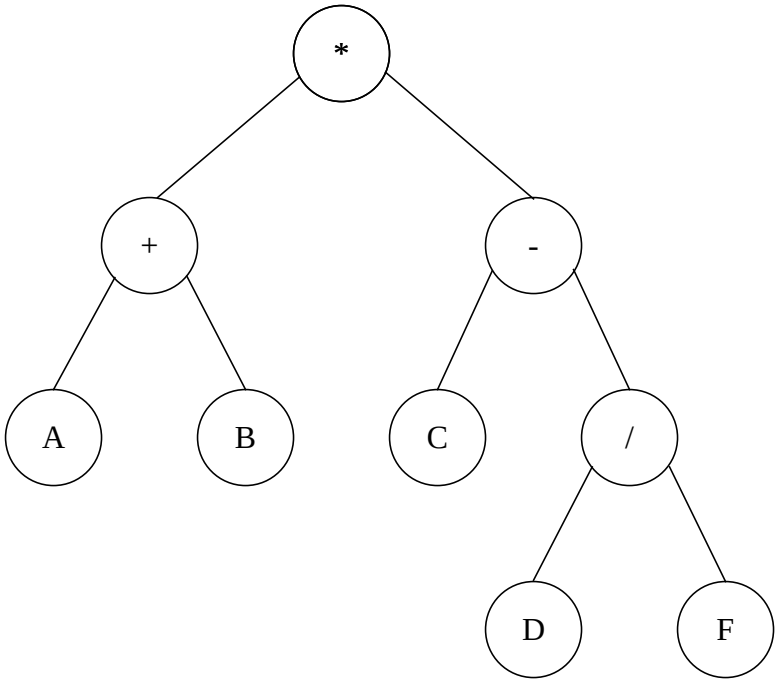


Figure 6: The expression tree for $(A + B) * (C - D/F)$.

either **Polish notation** (also called **prefix notation**) or to **reverse Polish notation** (also called **postfix notation**), as discussed below.

Let us first introduce some notation. As before, we write $\Theta(= \{+, -, *, **\})$ (here $**$ denotes the power operation) as an operand, while E_1 and E_2 are expression. The standard way of representing arithmetic expressions as shown above are called the infix notation. This can be written symbolically as $(E_1)\ \Theta\ (E_2)$. In the *prefix notation* (or Polish notation) we shall write

$$\Theta E_1 E_2$$

while in the *postfix notation* (or reverse Polish notation) we write

$$E_1 E_2 \Theta$$

Observe that parenthesis are *not* necessary. For the expression shown in (1) we have

$$\begin{aligned} \text{postfix notation} &= AB + CDF/ - *, \\ \text{prefix notation} &= * + AB - C/DF. \end{aligned}$$

How can we generate prefix and postfix notation from the infix notation. Actually, this is easy. We first build the expression tree, and then traverse it in preorder to get the prefix notation, and postorder to find the postfix notation. Indeed, consider the expression tree shown in Figure 6. The postorder traversal gives

$$AB + CDF/ - *$$

which agrees with the above. The preorder traversal leads us to

$$* + AB - C/DF$$

which is the same as above.

Exercise 8B: Write the following expression

$$A * B + C/D$$

in the postfix and prefix notations.

Theme 4: Graphs

In this section we present basic definitions and notations on graphs. As we mentioned in the Overview graphs are applied to solve various problems in computer science and engineering such as finding the shortest path between cities, building reliable computer networks, etc. We postpone an in-depth discussion of graphs to IT 320.

> A **graph** is a set of points (called **vertices**) together with a set of lines (called **edges**). There is at most one edge between any two vertices. More formally, a graph $G = (V, E)$ consists of a pair of sets V and E, where V is a set of vertices and $E \subset V \times V$ is the set of edges.

Example 6: In Figure 7 we present some graphs that will be used to illustrate our definitions. In particular, the first graph, say $G_1 = (V_1, E_1)$ has $V_1 = \{1, 2, 3, 4\}$ and $E_1 = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$. The second graph (that turns out to be a tree), say $G_2 = (V_2, E_2)$, consists of $V_2 = \{1, 2, 3, 4, 5, 6, 7\}$ and $E_2 = \{\{1, 2\}, \{1, 3\}, \{2, 4\}, \{2, 5\}, \{3, 6\}, \{3, 7\}\}$.

Now we list a number of useful notations associated with graphs.

- Two vertices are said to be **adjacent** if there is an edge between them. An edge $\{u, v\}$ is **incident** to vertices u and v. For example, in Figure 7 vertices $\{1\}$ and $\{2\}$ are adjacent, while the edge $\{2, 3\}$ is incident to $\{2\}$ and $\{3\}$.
- A **multigraph** has more than one edge between some vertices. Two edges between the same two vertices are said to be **parallel** edges.
- A **pseudograph** is a multigraph with loops. An edge is a **loop** if its start and end vertices are the same vertex.
- A **directed graph** or **digraph** has ordered pairs of directed edges. Each edge (v, w) has a start vertex v, and an end vertex w. For example, the last graph in Figure 7, $G_3 = (V_3, E_3)$, has $V_3 = \{1, 2, 3\}$ and the set of edges is $E_3 = \{(1, 2), (2, 1), (2, 3)\}$.

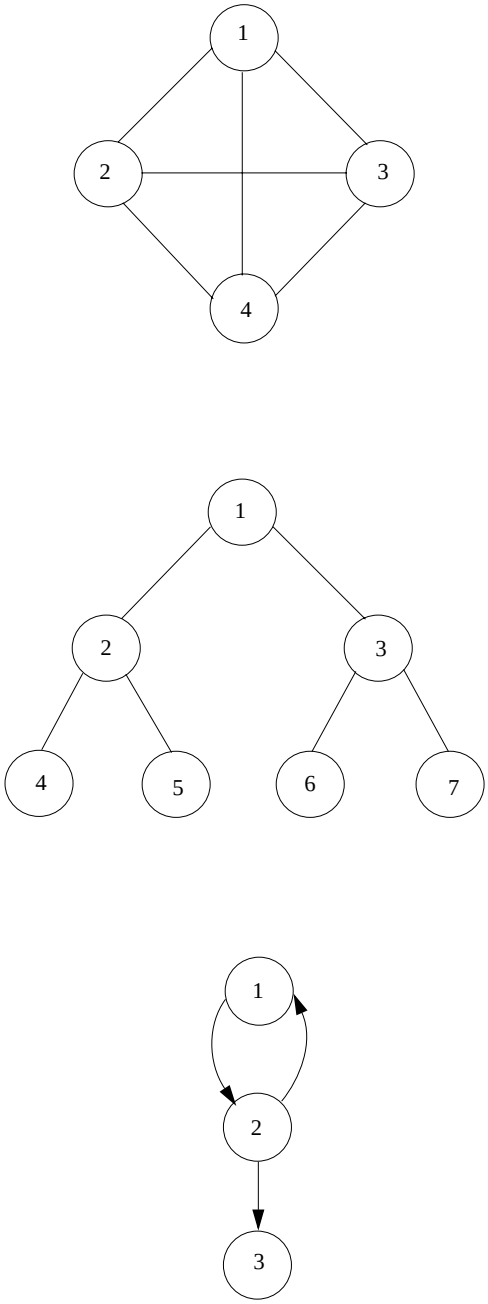
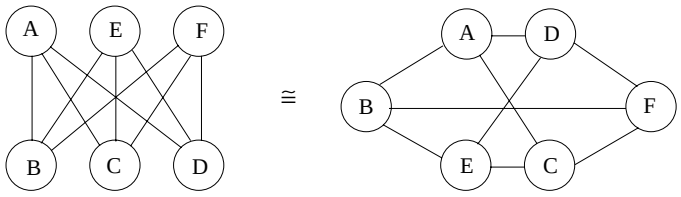


Figure 7: Examples of graphs.

- A **labeled** graph is a one-to-one and onto mapping of vertices to a set of unique labels, e.g., name of cities.

- Two graphs G and H are **isomorphic**, written $G \cong H$, iff there exists a one-to-one correspondence between their vertex sets which preserves adjacency. Thus



  are isomorphic since they have the same set of edges.

- A **subgraph** S of G is a graph having all vertices and edges in G; G is then a **supergraph** of S. That is, $S = (V_S, E_S)$ is a subgraph of $G = (V, E)$ if $V_S \subseteq V$ and $E_S \subseteq E$. A **spanning subgraph** is a subgraph containing all vertices of G, that is, $V_S = V$ and $E_S \subseteq E$. For example, n graph G_1 in Figure 7 the graph $S_1 = (V_1, E_1)$ with $V = \{1, 2\}$ and $E_1 = \{\{1, 2\}\}$ is a subgraph.

- If v_i is a vertex and if $n \geq 0$, we say that $(v_0, v_1, \ldots, v_n)$ is a **trail** if all edges are distinct, a **path** if all the vertices are distinct, and a *cycle* if the walk is a path and $v_0 = v_n$. The **length** is n. It must hold that if $0 \leq i < n$ then $(v_i, v_{i+1}) \in E$. In G_1 in Figure 7 $(\{1\}, \{3\}.\{4\})$ is a trail and a path.

- A graph is **connected** if there is a path between any two vertices of the graph. A vertex is **isolated** if there is no edge having it as one of its endpoints. Thus a connected graph has no isolated vertices. In Figure 7 graphs G_1 and G_2 are connected.

- The **girth** of a graph denoted by $g(G)$ is the length of the shortest cycle. In graph G_1 of Figure 7 we have $g(G_1) = 3$.

- The **circumference** of a graph denoted by $c(G)$, is the length of any longest cycle, and is undefined if no cycle of G exists. In graph G_1 of Figure 7 we have $c(G_1) = 4$.

- A graph is called **planar** if it can be drawn in the plane so that two edges, intersect only at points corresponding to nodes of the graph.

- Let $d(u, v)$ be the shortest length path between vertices u and v, if any. Then for all u, v, w in V:

  1. If $(u, v) \in E$ then $d(u, v) = d(v, u) = 1$.

2. $d(u,v) \leq 0$ with $d(u,v) = 0$ iff $u = v$.
3. $d(u,v) = d(v,u)$.
4. $d(u,v) + d(v,w) \geq d(u,w)$ {Triangular inequality}.

Thus, $d(u,v)$ defines a distance on graphs.

- A **degree** of a vertex v, denoted as $\deg(v)$ is the number of edges incident to v.

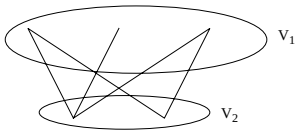
$$\sum_{v \in V} \deg(v) = 2|E|,$$

that is, the sum of vertex degrees is equal to twice the number of edges. The reader should verify it on Figure 7.

- A graph G is **regular** of degree r if every vertex has degree r. Graph G_1 in Figure 7 is 3-regular.

- A **complete** graph $K_n = (V, E)$ on n vertices has an edge between every pair of distinct vertices. Thus a complete graph K_n is regular degree of $n-1$, and has $n(n-1)/2$ edges. Observe that K_3 is a triangle. In Figure 7 $G_1 = K_4$.

- A **bipartite** graph, also refereed to as "bicolorable" or **bigraph**, is a graph whose vertex set can be separated into two disjoint sets such that there is no edge between two vertices of the same set. Thus a graph G is a bigraph if $G = (V_1 \cup V_2, E)$ such $V_1 \cap V_2 = \emptyset$ and for each edge (v,w) in E, either $v \in V_1$ and $w \in V_2$, or $v \in V_2$ and $w \in V_1$. A bigraph $K_{m,n}$ is such that $m = |V_1|$ and $n = |V_2|$.



- A **free tree** ("unrooted tree") is a connected graph with no cycles. G is a free tree if

1. G is connected, but if any edge is deleted the resulting graph is no longer connected.
2. If v and w are distinct vertices of G, then there is exactly one simple path from v to w.
3. G has no cycles and has $|V| - 1$ edges.

- A graph is **acyclic** if it contains no cycles.

- In a digraph the out-degree, denoted $d_{out}(v)$ of a vertex v is the number of edges with their initial vertex being v. Similarly the in-degree of a vertex v is the number of edges with their final vertex being v. Clearly for any digraph

$$\sum_{v \in V} d_{out}(v) = \sum_{v \in V} d_{in}(v),$$

that is, the sum of in-degrees over all vertices is the sum of out-degrees over all vertices (cf. Figure 7 for G_3). An acyclic digraph contains no directed cycles, and has at least one point of out-degree zero and at least on point of in-degree zero.

- A directed graph is said to be **strongly connected** if there is an oriented path from v to w and from w to v for any two vertices $v \neq w$. Graph G_3 in Figure 7 is *not* strongly connected since there is no path between vertex 3 and 1.

- If a graph contains a walk that traverses each edge exactly once, goes through all the vertices, and ends at the starting point, then the graph is said to be **Eulerian**. That is, it contains an **Eulerian trail**. None of the graphs in Figure 7 has an Eulerian trail.

- If there is a path through all vertices that visit every vertex once, then it is called a **Hamiltonian path**. If it ends in the starting point, then we have a **Hamiltonian cycle**. The Hamiltonian cycle in Figure 7 is $(\{1\}, \{2\}, \{3\}, \{4\})$.

- The **square** G^2 of a graph $G = (V, E)$ is $G^2(V, E')$ where E' contains an edge (u, v) whenever there is a path in G such that $d(u, v) \leq 2$. The powers G^3, $G^4, \ldots$ are defined similarity. Thus G^n is a graph which contains edges (u, v) between any two vertices that are connected by a path of length smaller than or equal to n in G. So $G^{|V|-1}$ is a graph which contains an edge between any two vertices that are connected by a path of any length in G. The graph $G^{|V|-1}$ is called the **transitive closure** of G.