Department of Mathematics, Government Arts College (Autonomous), Coimbatore-18

Year	Subject Title	Sem.	Sub Code
2018 -19 Onwards	DISCRETE MATHEMATICS FOR COMPUTER SCIENCE	II	18BCS24A

OBJECTIVES:

1. To understand the concepts of basic Discrete structures.
2. To get knowledge about applying the properties of Discrete Mathematical Structures.

UNIT: V

GRAPH THEORY: Basic terminology – Types of graphs – Paths, cycle and connectivity – Representation of graphs in computer memory – Trees – Properties of trees – Binary trees – Traversing binary trees – Computer representation of general trees.

(Chapter V - Sections: 5.1, 5.2)

5-1 BASIC CONCEPTS OF GRAPH THEORY

The terminology used in graph theory is not standard. It is not uncommon to find several different terms being used as synonyms. This situation, however, becomes more complicated when we find that a particular term is used by different authors to describe different concepts. This situation is natural because of the diversity of the fields in which graph theory is applied. Wherever possible, we shall indicate the alternative terms which are frequently used. We shall generally select alternatives that are often used in the literature in computer science.

In this section we shall define a graph as an abstract mathematical system. However, in order to provide some motivation for the terminology used and also to develop some intuitive feelings, we shall represent graphs diagrammatically. Any such diagram will also be called a graph. Our definitions and terms are not restricted to those graphs which can be represented by means of diagrams, even though this may appear to be the case because these terms have strong associations with such a representation. We shall see later on that a diagrammatic representation is only suitable in some very simple cases. Alternative methods of representing graphs will also be discussed. After introducing the terminology, we shall also discuss some of the basic results and theorems of graph theory.

5-1.1 Basic Definitions

Recall that in Chap. 2 a binary relation in a set V was defined as a subset of $V \times V$. It was shown that such a relation could be represented at least in some cases by a diagram which was called the graph of the relation. An alternative method of representing a relation was given by means of a relation or an incidence matrix. In this section we shall extend these ideas and in some ways generalize them.

We first consider several graphs which are represented by means of diagrams. Some of these graphs may be considered as graphs of certain relations, but there are others which cannot be interpreted in this manner.

Consider the diagrams shown in Fig. 5-1.1. For our purpose here, these diagrams represent graphs. Notice that every diagram consists of a set of points which are shown by dots or circles and are sometimes labeled $v_1, v_2, \ldots$, or $1, 2, \ldots$. Also in every diagram certain pairs of such points are connected by lines or arcs. The other details, such as the geometry of the arcs, their lengths, the position of the points etc., are of no importance at present. Notice that every arc starts at one point and ends at another point. A definition of the graph which is essentially an abstract mathematical system will now be given. Such a mathematical system is an abstraction of the graphs given in Fig. 5-1.1.

Definition 5-1.1 A *graph* $G = \langle V, E, \phi \rangle$ consists of a nonempty set V called the set of *nodes* (*points, vertices*) of the graph, E is said to be the set of *edges* of the graph, and ϕ is a mapping from the set of edges E to a set of ordered or unordered pairs of elements of V

We shall assume throughout that both the sets V and E of a graph are finite. It would be convenient to write a graph G as $\langle V, E \rangle$, or simply as G. Notice

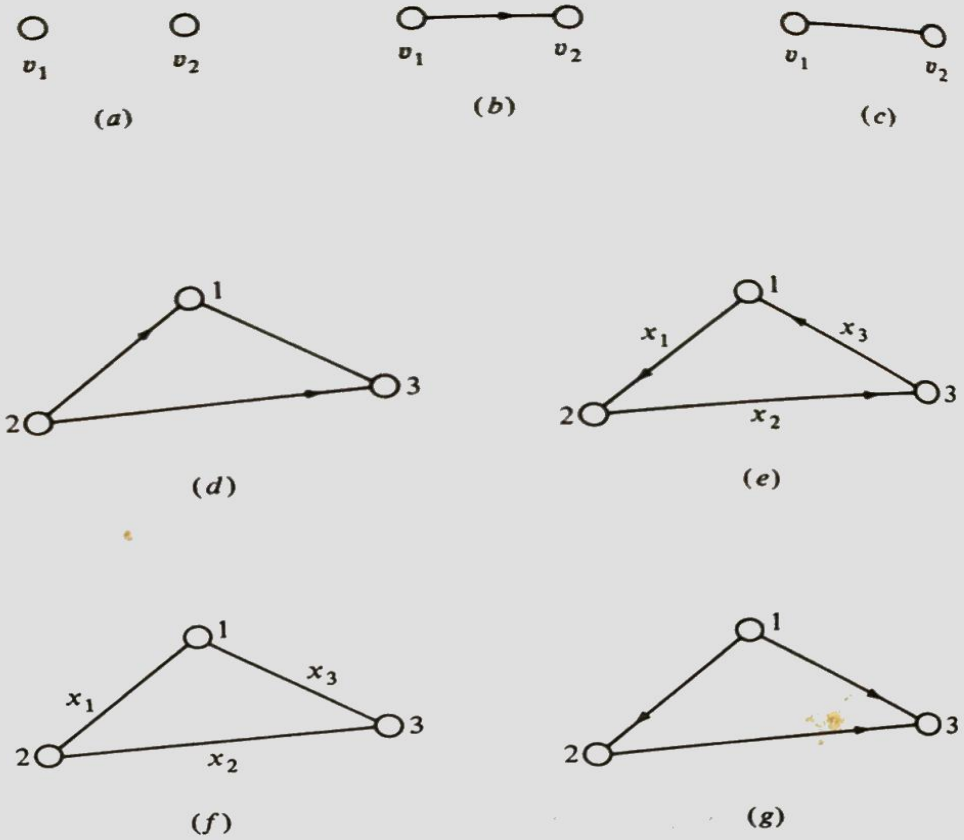


FIGURE 5-1.1

that the definition of a graph implies that to every edge of the graph G we can associate an ordered or unordered pair of nodes of the graph. If an edge $x \in E$ is thus associated with an ordered pair $\langle u, v \rangle$ or an unordered pair (u, v) where $u, v \in V$, then we say that the edge x connects or joins the nodes u and v. Any pair of nodes which are connected by an edge in a graph is called *adjacent* nodes.

Definition 5-1.2 In a graph $G = \langle V, E \rangle$, an edge which is associated with an ordered pair of $V \times V$ is called a *directed edge* of G, while an edge which is associated with an unordered pair of nodes is called an *undirected edge*. A graph in which every edge is directed is called a *digraph*, or a *directed graph*. A graph in which every edge is undirected is called an *undirected graph*. If some edges are directed and some are undirected in a graph, the graph is called *mixed*.

In the diagrams the directed edges are shown by means of arrows which also show the directions. The graphs given in Fig. 5-1.1*b*, *e*, and *g* are directed graphs. Those given in *c* and *f* are undirected, while the one given in *d* is mixed. The graph given in Fig. 5-1.1*a* could be considered as either directed or undirected. Notice that the edges x_1, x_2, and x_3 in Fig. 5-1.1*e* are associated with the ordered pairs $\langle 1, 2 \rangle$, $\langle 2, 3 \rangle$, and $\langle 3, 1 \rangle$ respectively, while the edges x_1, x_2, and x_3 in *f* are associated with the unordered pairs $(1, 2)$, $(2, 3)$, and $(3, 1)$ respectively. In Fig. 5-1.1*f* the node 1 is adjacent to nodes 2 and 3.

A city map showing only the one-way streets is an example of a directed

graph in which the nodes are the intersections and the edges are the streets. A map showing only the two-way streets is an example of an undirected graph, while a map showing all the one-way and two-way streets is an example of a mixed graph.

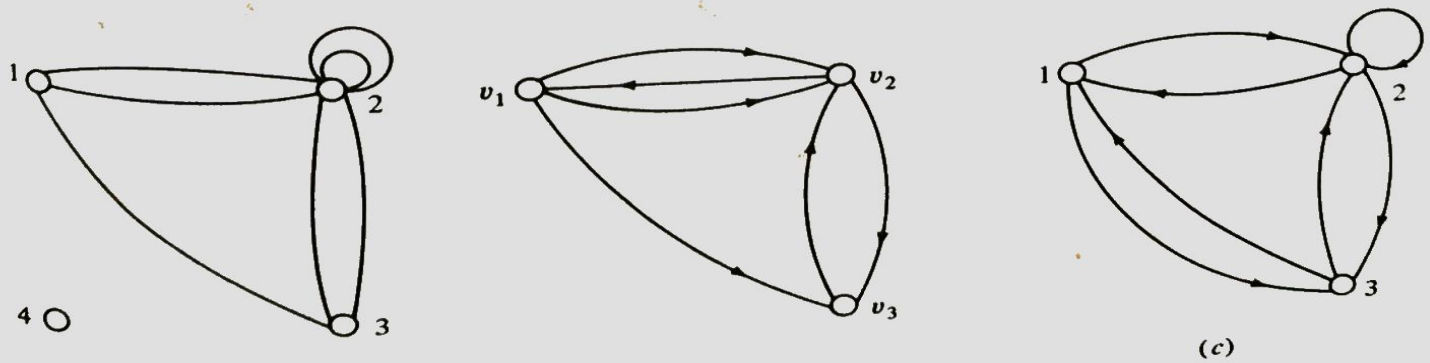
Let $\langle V, E \rangle$ be a graph and let $x \in E$ be a directed edge associated with the ordered pair of nodes $\langle u, v \rangle$. Then the edge x is said to be *initiating* or *originating* in the node u and *terminating* or *ending* in the node v. The nodes u and v are also called the *initial* and *terminal* nodes of the edge x. An edge $x \in E$ which joins the nodes u and v, whether it be directed or undirected, is said to be *incident* to the nodes u and v.

An edge of a graph which joins a node to itself is called a *loop* (*sling*) (not to be confused with a loop in a program). The direction of a loop is of no significance; hence it can be considered either a directed or an undirected edge. Some authors do not admit any loops in the definition of a graph. We shall see that the presence or absence of a loop does not significantly change the theorems of graph theory.

The graphs given in Fig. 5-1.1 have no more than one edge between any pair of nodes. In the case of directed edges, the two possible edges between a pair of nodes which are opposite in direction are considered distinct. In some directed as well as undirected graphs, we may have certain pairs of nodes joined by more than one edge, as shown in Fig. 5-1.2*a* and *b*. Such edges are called *parallel*. Note that there are no multiple edges in the graph of Fig. 5-1.2*c*. In Fig. 5-1.2*a* there are two parallel edges joining the nodes 1 and 2, two parallel edges joining the nodes 2 and 3, while there are two parallel loops at 2. In 5-1.2*b* there are two parallel edges associated with the ordered pair $\langle v_1, v_2 \rangle$.

Any graph which contains some parallel edges is called a *multigraph*. On the other hand, if there is no more than one edge between a pair of nodes (no more than one directed edge in the case of a directed graph), then such a graph is called *a simple graph*. The graphs given in Fig. 5-1.1 are all simple graphs.

The graphs in Fig. 5-1.2*a* and *b* may be represented by the diagrams given in Fig. 5-1.3*a* and *b* in which the number on any edge shows the multiplicity of the edge. We may also consider the multiplicity as a weight assigned to an edge. This interpretation allows us to generalize the concept of weight to numbers which are not necessarily integers. We may thus have graphs as shown in Fig.



(*a*) (*b*) (*c*)

FIGURE 5-1.2

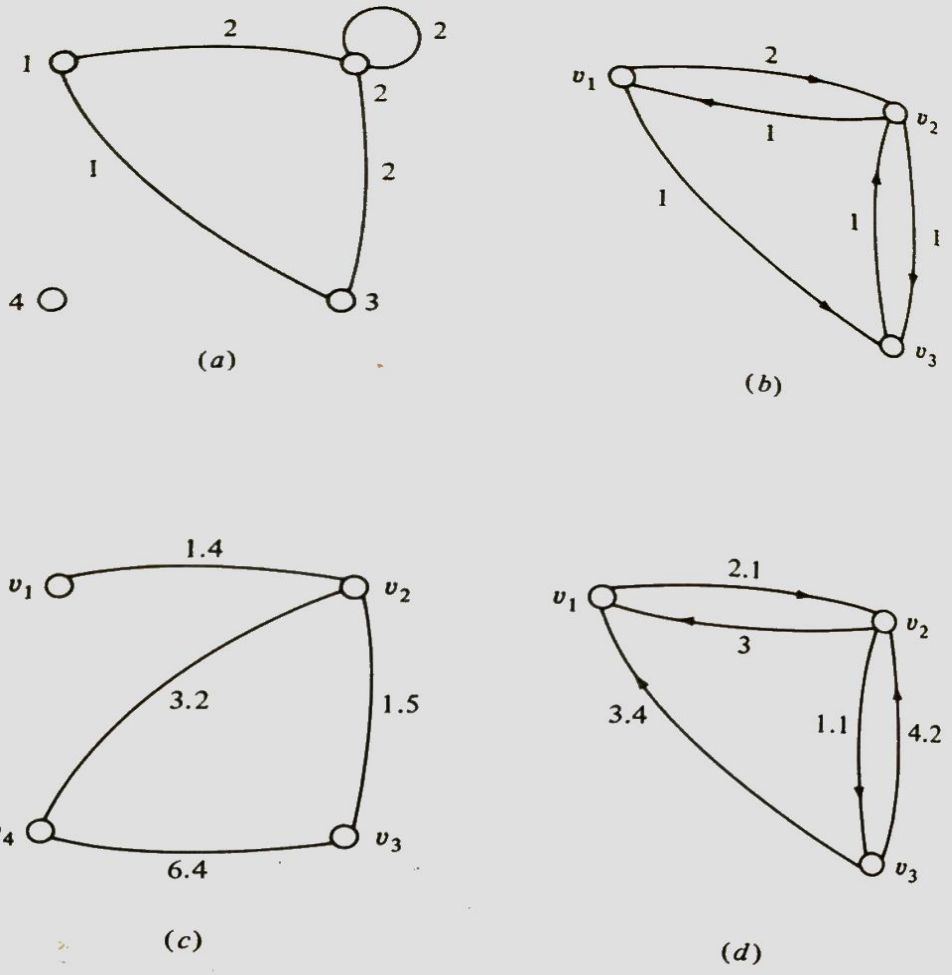


FIGURE 5-1.3

5-1.3*c* and *d* in which the numbers on the edges show the weights of the edges. A graph in which weights are assigned to every edge is called a *weighted graph.*

A graph representing a system of pipelines in which the weights assigned indicate the amount of some commodity transferred through the pipe is an example of a weighted graph. Similarly, a graph of city streets may be assigned weights according to the traffic density on each street.

In a graph a node which is not adjacent to any other node is called an *isolated node.* A graph containing only isolated nodes is called a *null graph.* In other words, the set of edges in a null graph is empty. The graph in Fig. 5-1.1*a* is a null graph, while that in Fig. 5-1.2*a* has an isolated node. In practice, an isolated node in a graph has very little importance.

The definition of graph contains no reference to the length or the shape and positioning of the arc joining any pair of nodes, nor does it prescribe any ordering of positions of the nodes. Therefore, for a given graph, there is no unique diagram which represents the graph. We can obtain a variety of diagrams by locating the nodes in an arbitrary number of different positions and also by showing the edges by arcs or lines of different shapes. Because of this arbitrariness, it can happen that two diagrams which look entirely different from one another may represent the same graph, as in Fig. 5-1.4*a*, *a'*, also in *b*, *b'*, and *c*, *c'*. The graphs in *c* and *c'* need further explanation, because the nodes are also labeled differently in these two graphs.

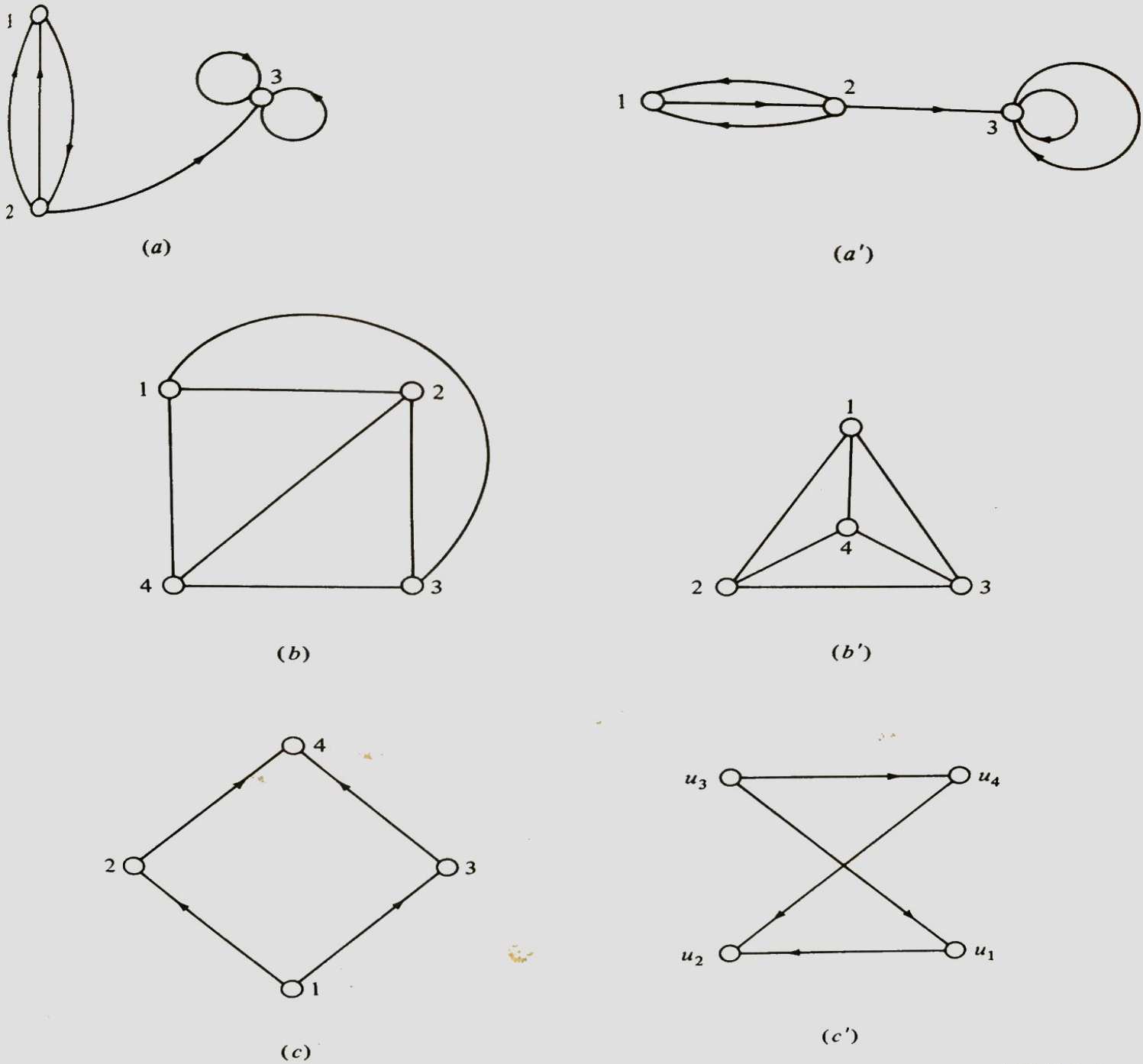


FIGURE 5-1.4

Definition 5-1.3 Two graphs are *isomorphic* if there exists a one-to-one correspondence between the nodes of the two graphs which preserves adjacency of the nodes as well as the directions of the edges, if any.

According to the definition of isomorphism we note that any two nodes in one graph which are joined by an edge must have the corresponding nodes in the other graph also joined by an edge, and hence a one-to-one correspondence exists between the edges as well. The graphs given in Fig. 5-1.4*c* and *c′* are isomorphic because of the existence of a mapping

$$1 \longrightarrow u_3 \qquad 2 \longrightarrow u_1 \qquad 3 \longrightarrow u_4 \qquad \text{and} \qquad 4 \longrightarrow u_2$$

Under this mapping, the edges $\langle 1, 3\rangle$, $\langle 1, 2\rangle$, $\langle 2, 4\rangle$, and $\langle 3, 4\rangle$ are mapped into $\langle u_3, u_4\rangle$, $\langle u_3, u_1\rangle$, $\langle u_1, u_2\rangle$, and $\langle u_4, u_2\rangle$ which are the only edges of the graph in c'.

Note that it is essential that two graphs which are isomorphic have the same number of nodes and edges; however, this is not a sufficient condition for an isomorphism to exist, as can be seen from the graphs given in Fig. 5-1.1*e* and *g* which are not isomorphic.

Definition 5-1.4 In a directed graph, for any node v the number of edges which have v as their initial node is called the *outdegree* of the node v. The number of edges which have v as their terminal node is called the *indegree* of v, and the sum of the outdegree and the indegree of a node v is called its *total degree*. In case of an undirected graph, the *total degree* or the *degree* of a node v is equal to the number of edges incident with v.

The total degree of an isolated node is 0.

The concept of the degree of a node can be generalized to a set of nodes. Let $G = \langle V, E\rangle$ be a directed graph and $X \subseteq V$ be a subset of nodes. The number of edges of G which have their initial node in X but their terminal node not in X is called the *outdegree* of X. Similarly, the number of edges of G which have their terminal nodes in X but their initial nodes not in X is called the *indegree* of X. The indegree and outdegree of a node are a special case of this definition.

A simple result involving the notion of the degree of nodes of a graph is that the sum of the degrees (or total degrees in the case of a directed graph) of all the nodes of a graph must be an even number which is equal to twice the number of edges in the graph.

Let $V(H)$ be the set of nodes of a graph H and $V(G)$ be the set of nodes of a graph G such that $V(H) \subseteq V(G)$. If, in addition, every edge of H is also an edge of G, then the graph H is called a *subgraph* of the graph G, which is expressed by writing $H \subseteq G$. Naturally, the graph G itself, as well as the null graph obtained from G by deleting all the edges of G, is also a subgraph of G. Other subgraphs of G can be obtained by deleting certain modes and edges of G.

We shall end this section by showing how the theory of binary relations given in Chap. 2 is closely linked to the theory of simple digraphs.

Let $G = \langle V, E\rangle$ be a simple digraph. Then every edge of E can be expressed by means of an ordered pair of elements of V. Such an ordered pair uniquely defines the edge; hence $E \subseteq V \times V$. On the other hand, any subset of $V \times V$ defines a relation in V. Accordingly, E is a binary relation in V whose graph is the same as the simple digraph G. This observation permits us to carry over the terminology and the results developed in Chap. 2 on binary relations.

A simple digraph $G = \langle V, E\rangle$ is called *reflexive*, *transitive*, *symmetric*, *antisymmetric*, etc., if the relation E is reflexive, transitive, symmetric, antisymmetric, etc. We can also define the *converse of a digraph* $G = \langle V, E\rangle$ to be a digraph $\tilde{G} = \langle V, \tilde{E}\rangle$ in which the relation $\tilde{E}$ is the converse of the relation E. The diagram of $\tilde{G}$ is obtained from that of G by simply reversing the directions of the edges in G. The converse $\tilde{G}$ is also called the *reversal* or *directional dual* of a digraph G.

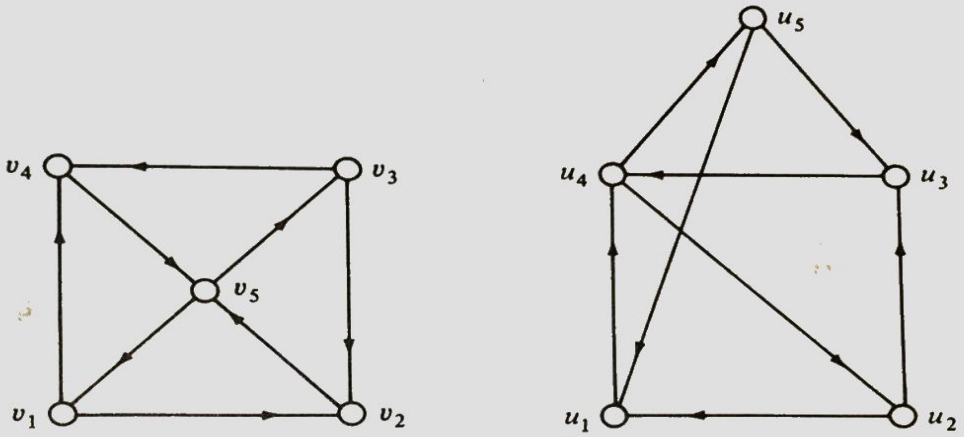
If a simple digraph $G = \langle V, E\rangle$ is reflexive, symmetric, and transitive, then the relation E must be an equivalence relation on V, and hence V can be partitioned into equivalence classes. If we consider any such equivalence class of

nodes along with the edges that join these nodes, we have subgraphs of G. Thes
subgraphs are such that every node in the subgraph is adjacent to every othe
node of the subgraph. However, no node of any one subgraph is adjacent to any
node of another subgraph. In this sense the graph G is partitioned into subgraphs
which are disjoint.

EXERCISES 5-1.1

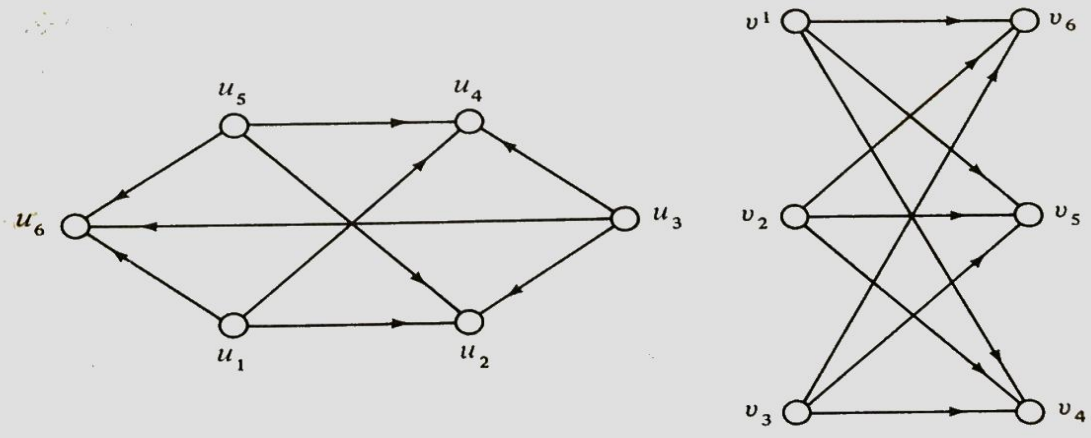
1 Show that the sum of indegrees of all the nodes of a simple digraph is equal to the sum of outdegrees of all its nodes and that this sum is equal to the number of edges of the graph.

2 Draw all possible simple digraphs having three nodes. Show that there is only one digraph with no edges, one with one edge, four with two edges, four with three edges, four with four edges, one with five edges, and one with six edges. Assume that there are no loops and that graphs which are isomorphic are not distinguishable.



(a) (b)

FIGURE 5-1.5



(a) (b)

FIGURE 5-1.6

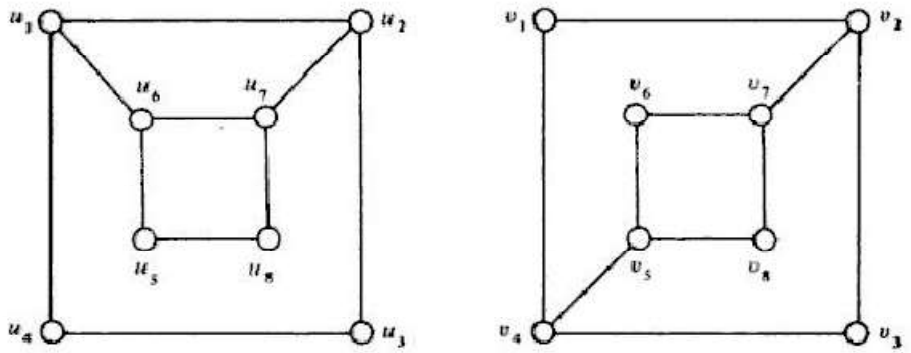


FIGURE 5-1.7

State the properties of these digraphs such as symmetry, transitivity, antisymmetry, etc.

3 Show that the digraphs given in Fig. 5-1.5*a* and *b* are isomorphic.

4 Show that the digraphs given in Fig. 5-1.6*a* and *b* are isomorphic.

5 Show that the digraphs in Fig. 5-1.7 are not isomorphic.

6 A simple digraph $G = \langle V, E \rangle$ is said to be *complete* if every node is adjacent to all other nodes of the graph. Show that a complete digraph with n nodes has the maximum number of edges viz., $n(n-1)$ edges, assuming that there are no loops.

7 The *complement* of a simple digraph $G = \langle V, E \rangle$ is the digraph $\bar{G} = \langle V, \bar{E} \rangle$ where $\bar{E} = V \times V - E$. Find the complements of the graphs given in Prob. 2.

5-1.2 Paths, Reachability, and Connectedness

In this section we introduce some additional terminology associated with a simple digraph. During the course of our discussion we shall also indicate how the same terminology and concepts can be extended to simple undirected graphs as well as to multigraphs.

Let $G = \langle V, E \rangle$ be a simple digraph. Consider a sequence of edges of G such that the terminal node of any edge in the sequence is the initial node of the next edge, if any, in the sequence. An example of such a sequence is

$$(\langle v_{i_1}, v_{i_2} \rangle, \langle v_{i_2}, v_{i_3} \rangle, \ldots, \langle v_{i_{k-2}}, v_{i_{k-1}} \rangle, \langle v_{i_{k-1}}, v_{i_k} \rangle)$$

where it is assumed that all the nodes and edges appearing in the sequence are in V and E respectively. It is customary to write such a sequence as

$$(v_{i_1}, v_{i_2}, \ldots, v_{i_{k-1}}, v_{i_k})$$

Note that not all edges and nodes appearing in a sequence need be distinct. Also, for a given graph any arbitrary set of nodes written in any order do not give a sequence as required. In fact each node appearing in the sequence must be adjacent to the nodes appearing just before and after it in the sequence, except in the case of the first and last nodes.

Definition 5-1.5 Any sequence of edges of a digraph such that the terminal node of any edge in the sequence is the initial node of the edge, if any, appearing next in the sequence defines a *path* of the graph.

A path is said to *traverse* through the nodes appearing in the sequence, *orig-*

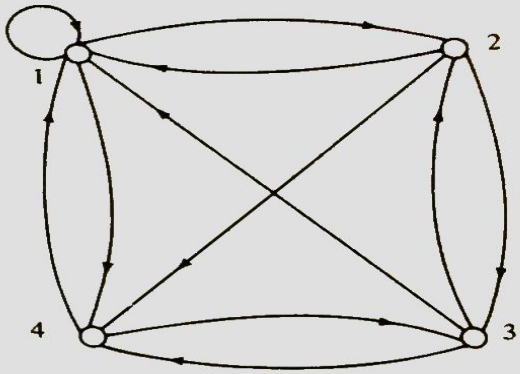


FIGURE 5-1.8

inating in the initial node of the first edge and *ending* in the terminal node of the last edge in the sequence.

Definition 5-1.6 The number of edges appearing in the sequence of a path is called the *length* of the path.

Consider the simple digraph given in Fig. 5-1.8. Some of the paths originating in node 1 and ending in node 3 are

$$P_1 = (\langle 1, 2\rangle, \langle 2, 3\rangle)$$
$$P_2 = (\langle 1, 4\rangle, \langle 4, 3\rangle)$$
$$P_3 = (\langle 1, 2\rangle, \langle 2, 4\rangle, \langle 4, 3\rangle)$$
$$P_4 = (\langle 1, 2\rangle, \langle 2, 4\rangle, \langle 4, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle)$$
$$P_5 = (\langle 1, 2\rangle, \langle 2, 4\rangle, \langle 4, 1\rangle, \langle 1, 4\rangle, \langle 4, 3\rangle)$$
$$P_6 = (\langle 1, 1\rangle, \langle 1, 1\rangle, \ldots, \langle 1, 2\rangle, \langle 2, 3\rangle)$$

Definition 5-1.7 A path in a digraph in which the edges are all distinct is called a *simple path* (*edge simple*). A path in which all the nodes through which it traverses are distinct is called an *elementary path* (*node simple*).

Naturally, every elementary path of a digraph is also simple. The paths P_1, P_2, and P_3 of the digraph in Fig. 5-1.8 are elementary, while the path P_5 is simple but not elementary. We shall show here that if there exists a path from a node say, u, to another node v, then there must also be an elementary path from u to v.

Definition 5-1.8 A path which originates and ends in the same node is called a *cycle* (*circuit*). A cycle is called *simple* if its path is simple, i.e., no edge in the cycle appears more than once in the path. A cycle is called *elementary* if it does not traverse through any node more than once.

Note that in a cycle the initial node appears at least twice even if it is an

elementary cycle. The following are some of the cycles in the graph of Fig. 5-1.8:

$$C_1 = (\langle 1, 1\rangle)$$
$$C_2 = (\langle 1, 2\rangle, \langle 2, 1\rangle)$$
$$C_3 = (\langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3, 1\rangle)$$
$$C_4 = (\langle 1, 4\rangle, \langle 4, 3\rangle, \langle 3, 1\rangle)$$
$$C_5 = (\langle 1, 4\rangle, \langle 4, 3\rangle, \langle 3, 2\rangle, \langle 2, 1\rangle)$$

Observe that any path which is not elementary contains cycles traversing through those nodes which appear more than once in the path. By deleting such cycles one can obtain elementary paths. For example, in the path P_5, if we delete the cycle $(\langle 1, 2\rangle, \langle 2, 4\rangle, \langle 4, 1\rangle)$, we obtain the path P_2, which also originates at 1 and ends in 3 and is an elementary path. Similarly, if in the path P_4 we delete the same cycle, we get the elementary path P_1. Likewise, it is possible to obtain elementary cycles at any node from a cycle at that node. Because of this property, some authors use the term "path" to mean only the elementary paths, and they likewise apply the notion of the length of a path to only elementary paths.

A simple digraph which does not have any cycles is called *acyclic*. Naturally, such graphs cannot have any loops. We consider a class of digraphs which are acyclic in Sec. 5-1.4.

Definition 5-1.9 A node v of a simple digraph is said to be *reachable* (*accessible*) from the node u of the same digraph, if there exists a path from u to v.

Note that the concept of reachability is independent of the number of alternate paths from u to v and also of their lengths. For the graph given in Fig. 5-1.8, we have given paths P_1 to P_5 from node 1 to node 3. Any one of these paths is sufficient to establish the reachability of node 3 from node 1. For the sake of completeness we shall assume that every node is reachable from itself.

If a node v is reachable from the node u, then a path of minimum length from u to v is called a *geodesic*. The length of a geodesic from the node u to the node v is called the *distance* and is denoted by $d\langle u, v\rangle$. It is assumed that $d\langle u, u\rangle = 0$ for any node u.

It is clear from the definition that reachability is a binary relation on the set of nodes of a simple digraph. By our definition, reachability is reflexive. If there exists a path from a node u to a node v, and a path from the node v to a node w, then clearly there is a path from u to w. In other words, reachability is also a transitive relation. In general, it is not true that if there is a path from u to v, then there exists a path from v to u. Therefore, reachability is not necessarily symmetric, nor is it antisymmetric.

The distance from a node u to a node v, if v is reachable from u, is written as $d\ u, v$ and satisfies the following properties:

$$d\langle u, v\rangle \geq 0$$
$$d\langle u, u\rangle = 0$$
$$d\langle u, v\rangle + d\langle v, w\rangle \geq d\langle u, w\rangle$$

The last inequality is called the *triangle inequality*. If v is not reachable from u, then it is customary to write $d\langle u, v\rangle = \infty$. Note also that if v is reachable from u and u is reachable from v, then $d\langle u, v\rangle$ is not necessarily equal to $d\langle v, u\rangle$.

The following theorem about the length of an elementary path between two nodes of a simple digraph will be used in Sec. 5-1.3.

Theorem 5-1.1 In a simple digraph, the length of any elementary path is less than or equal to $n - 1$, where n is the number of nodes in the graph. Similarly, the length of any elementary cycle does not exceed n.

PROOF The proof is based upon the fact that in any elementary path the nodes appearing in the sequence are distinct. The number of distinct nodes in any elementary path of length k is $k + 1$. Since there are only n distinct nodes in the graph, we cannot have an elementary path of length greater than $n - 1$. For an elementary cycle of length k, the sequence contains k distinct nodes; hence the result. ////

Let us now briefly consider how the concepts of path and cycle can be extended to undirected graphs. Notice that the definition of a path requires that the edges appearing in the sequence must have a definite initial and terminal node. In the case of a simple undirected graph, an edge is given by an unordered pair, and any one of the nodes in the ordered pair can be considered as the initial or the terminal node of the edge. In order to apply the same definition of a path to an undirected graph, we consider every edge in an undirected graph to be replaced by two directed edges in opposite directions. Once this is done, we have a directed graph, and all the definitions of path, cycle, elementary path, reachability, etc., are carried over to undirected graphs. In the case of an undirected graph, the reachability relation is symmetric and so also is the distance. Theorem 5-1.1 holds for undirected graphs.

For a directed graph $G = \langle V, E\rangle$ we shall now extend the concept of reachability of a node. The set of nodes which are reachable from a given node v is said to be the *reachable set* of v. The reachable set of v is written as $R(v)$. For any subset $S \subseteq V$, the *reachable set* of S is the set of nodes which are reachable from any node of S, and this set is denoted by $R(S)$.

For the digraph given in Fig. 5-1.9, some of the reachable sets are as follows:

$$R(v_1) = \{v_1, v_2, v_3, v_4, v_5, v_6\} = R(v_2) = R(v_3) = R(v_4) = R(v_5)$$

$$R(v_6) = \{v_6\} \qquad R(v_7) = \{v_6, v_7\} \qquad R(v_8) = \{v_6, v_7, v_8\}$$

$$R(v_9) = \{v_9\} \qquad R(v_{10}) = \{v_{10}\} \qquad R(v_5, v_8, v_9, v_{10}) = V = R(v_1, v_8, v_9, v_{10})$$

In a digraph $G = \langle V, E\rangle$, a subset $X \subseteq V$ is called a *node base* if its reachable set is V and if no proper subset of X has this property.

In the digraph of Fig. 5-1.9, the set $\{v_1, v_8, v_9, v_{10}\}$ is a node base and so is the set $\{v_5, v_8, v_9, v_{10}\}$. There are several interesting facts about a node base of a simple digraph. We shall mention here a few of these facts, but we shall not prove these statements for the proofs are quite simple and straightforward.

Every isolated point of a digraph must be present in a node base. Any node whose indegree is zero must be present in any node base. From the definition

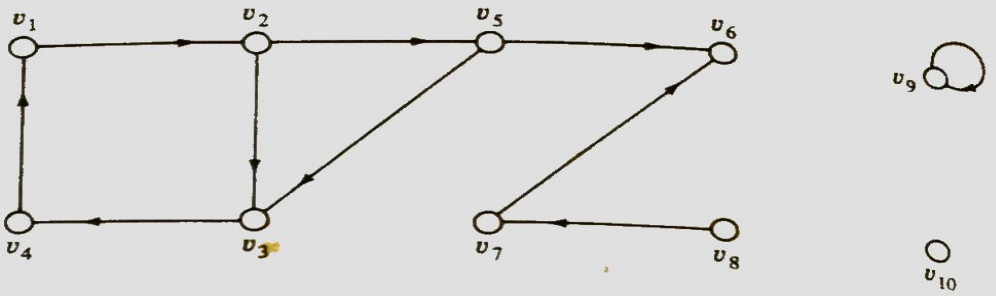


FIGURE 5-1.9

of a node base, it is clear that no node in the node base is reachable from another node in the node base. From the nodes lying on an elementary cycle any one node could be chosen as an element of a node base. Consequently, any node that does not have indegree zero and that does not lie on a cycle cannot be present in a node base. In a acyclic graph, a node base consists of only those nodes whose indegree is zero.

For a given simple digraph, we may have more than one node base; however, every node base has the same number of nodes. This statement is proved by showing that for any two node bases S_1 and S_2, every node of S_2 is reachable from exactly one node of S_1, and conversely. Thus a one-to-one correspondence is established between S_1 and S_2.

We shall now introduce an important concept, viz., that of the connectedness of nodes in a graph.

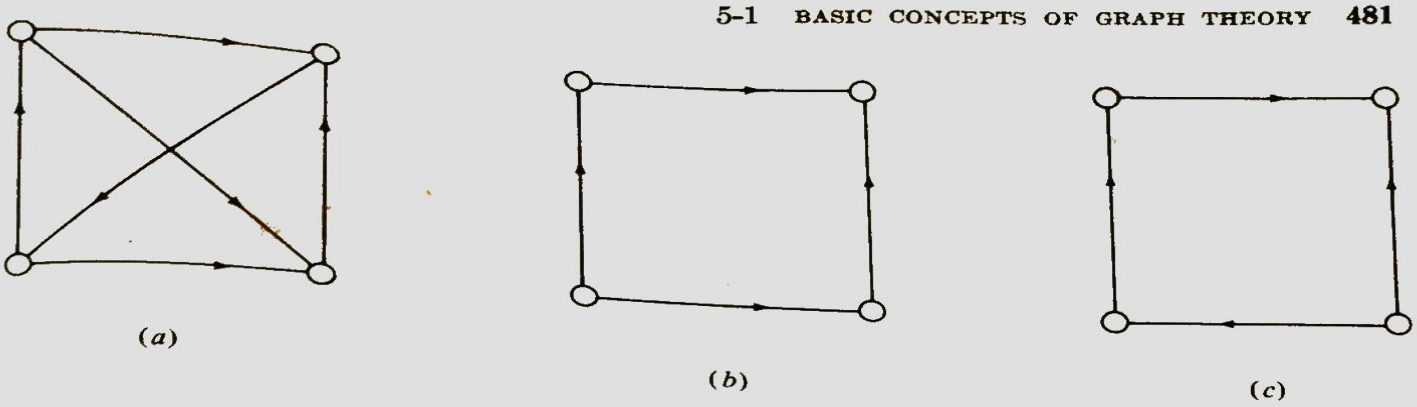
An undirected graph is said to be *connected* if for any pair of nodes of the graph the two nodes are reachable from one another. This definition cannot be applied to directed graphs without some further modifications, because in a directed graph if a node u is reachable from another node v, the node v may not be reachable from u. In order to overcome this difficulty, we call a digraph *connected* (*weakly connected*) if it is connected as an undirected graph in which the direction of the edges is neglected, i.e., if the graph when treated as an undirected graph is connected.

Definition 5-1.10 A simple digraph is said to be *unilaterally connected* if for any pair of nodes of the graph at least one of the nodes of the pair is reachable from the other node. If for any pair of nodes of the graph both the nodes of the pair are reachable from one another, then the graph is called *strongly connected*.

Observe that a unilaterally connected digraph is weakly connected, but a weakly connected digraph is not necessarily unilaterally connected. In fact, in a weakly connected digraph we may find that for any pair of nodes, say u and v, neither u is reachable from v nor v is reachable from u. A strongly connected digraph is both unilaterally and weakly connected.

For the digraphs given in Fig. 5-1.10, the digraph in Fig. 5-1.10*a* is strongly connected, *b* is weakly connected but not unilaterally connected, while *c* is unilaterally connected but not strongly connected.

Let $G = \langle V, E \rangle$ be a simple digraph and $X \subseteq V$. A subgraph whose nodes



(a) (b) (c)

FIGURE 5-1.10

are given by the set X and whose edges consist of all those edges of G which have their initial and terminal nodes in X is called the *subgraph generated by* X. A subgraph G_1 is said to be maximal with respect to some property if no other subgraph has the property and also includes G_1.

Definition 5-1.11 For a simple digraph, a maximal strongly connected subgraph is called a *strong component*. Similarly, a maximal unilaterally connected or maximal weakly connected subgraph is called a *unilateral* or *weak component* respectively.

For the digraph given in Fig. 5-1.11, {1, 2, 3}, {4}, {5}, {6} are the strong components. {1, 2, 3, 4, 5}, {6} are the unilateral components, and {1, 2, 3, 4, 5, 6} is the weak component because the graph is weakly connected.

Theorem 5-1.2 In a simple digraph, $G = \langle V, E \rangle$, every node of the digraph lies in exactly one strong component.

PROOF Let $v \in V$ and S be the set of all those nodes of G which are mutually reachable with v. The set S naturally contains v and is a strong component of G. This shows that every node of G is contained in a strong component.

Assume now that a node v is in two strong components. It would imply that any node in one strong component which contains v is reachable from any node in the other strong component which also contains v, because every such path is easily established through v. This, however, is impossible. Hence every node is contained in exactly one strong component. Thus the strong components partition V. ////

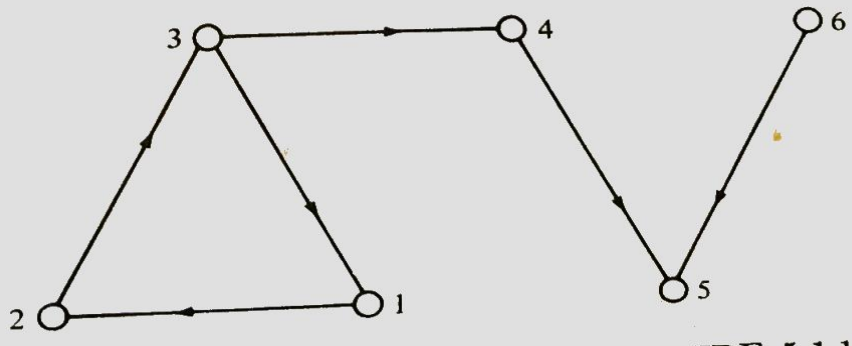


FIGURE 5-1.11

Note that any edge $x \in E$ of a simple digraph may or may not be contained in a strong component. If $x = \langle u, v \rangle$ and both u and v are in a strong component S, then x is in a strong component. If an edge $x \in E$ is in a strong component, then x must be a part of a cycle because if $\langle u, v \rangle \in S$, then $\langle v, u \rangle$ is also in S.

Similar results can be proved for weak and unilateral components. We shall simply state these results. The proof is similar to the one given for Theorem 5-1.2.

Every node and edge of a simple digraph is contained in exactly one weak component.

Every node and edge of a simple digraph lies in at least one unilateral component.

We shall now show how a simple digraph can be used to represent the resource allocation status of an operating system.

In a multiprogrammed computer system, it appears that several programs are executed at one time. In reality, the programs are sharing the resources of the computer system, such as tape units, disc devices, the central processor, main memory, and compilers. A special set of programs called an operating system controls the allocation of these resources to the programs. When a program requires the use of a certain resource, it issues a request for that resource, and the operating system must ensure that the request is satisfied.

It may happen that requests for resources are in conflict. For example, program A may have control of resource r_1 and require resource r_2, but program B has control of resource r_2 and requires resource r_1. In such a case, the computer system is said to be in a state known as *deadlock*, and the conflicting requests must be resolved. A directed graph can be used to model resource requests and assist in the detection and correction of deadlocks.

It is assumed that all resource requests of a program must be satisfied before that program can complete execution. If any requested resources are unavailable at the time of the request, the program will assume control of the resources which are available, but must wait for the unavailable resources.

Let $P_t = \{p_1, p_2, \ldots, p_m\}$ represent the set of programs in the computer system at time t. Let $A_t \subseteq P_t$ be the set of active programs, or programs that have been allocated at least a portion of their resource requests at time t. Finally, let $R_t = \{r_1, r_2, \ldots, r_n\}$ represent the set of resources in the system at time t. An allocation graph G_t is a directed graph representing the resource allocation status of the system at time t and consisting of a set of nodes $V = R_t$ and a set of edges E. Each resource is represented by a node of the graph. There is a directed edge from node r_i to r_j if and only if there is a program p_k in A_t that has been allocated resource r_i but is waiting for r_j.

For example, let $R_t = \{r_1, r_2, r_3, r_4\}$, $A_t = \{p_1, p_2, p_3, p_4\}$, and the resource allocation status be

p_1 has resource r_4 and requires r_1

p_2 has resource r_1 and requires r_2 and r_3

p_3 has resource r_2 and requires r_3

p_4 has resource r_3 and requires r_1 and r_4

Then the allocation graph at time t is given in Fig. 5-1.12.

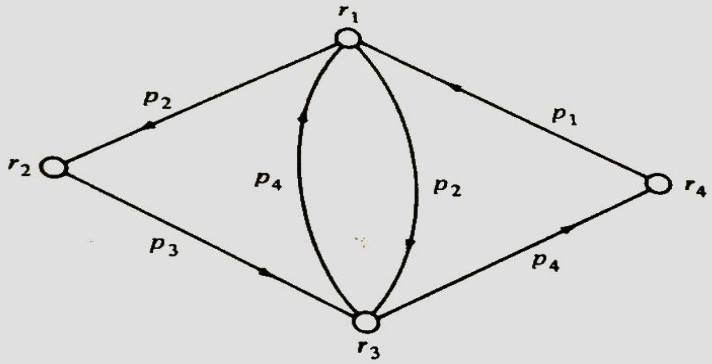


FIGURE 5-1.12 Allocation graph for detecting deadlocks.

It can be shown that the state of deadlock exists in a computer system at time t if and only if the allocation graph G_t contains strongly connected components. In the case of our example, the graph G_t is strongly connected. In fact, in Sec. 5-1.3 we discuss methods which will identify the nodes in a strong component and thus detect the resources and programs which are involved in the deadlock.

EXERCISES 5-1.2

1 Give three different elementary paths from v_1 to v_3 for the digraph given in Fig. 5-1.13. What is the shortest distance between v_1 and v_3? Is there any cycle in the graph? Is the digraph transitive? In case it is not transitive, find the transitive closure (see Sec. 2-3.7) of the digraph.

2 Find all the indegrees and outdegrees of the nodes of the graph given in Fig. 5-1.14. Give all the elementary cycles of this graph. Obtain an acyclic digraph by deleting one edge of the given digraph. List all the nodes which are reachable from another node of the digraph.

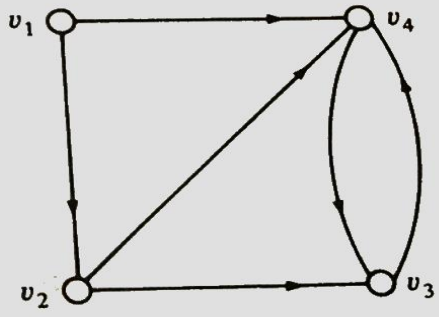


FIGURE 5-1.13

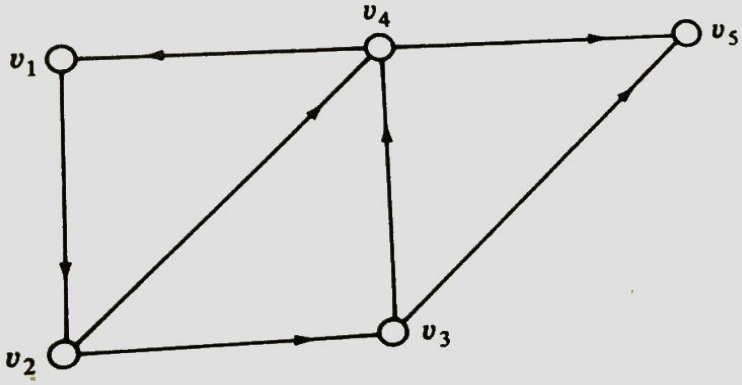


FIGURE 5-1.14

3 Given a simple digraph $G = \langle V, E \rangle$, under what condition is the equation

$$d(v_1, v_2) + d(v_2, v_3) = d(v_1, v_3)$$

satisfied for v_1, v_2, and $v_3 \in V$?

4 Find the reachable sets of $\{v_1, v_4\}$, $\{v_4, v_5\}$, and $\{v_3\}$ for the digraph given in Fig. 5-1.14.

5 Find a node base for each of the digraphs given in Figs. 5-1.13 and 5-1.14.

6 Explain why no node in a node base is reachable from another node in the node base.

7 Prove that in an acyclic simple digraph a node base consists of only those nodes whose indegree is zero.

8 For the digraphs given in Figs. 5-1.13 and 5-1.14, determine whether they are strongly, weakly, or unilaterally connected.

9 Show that a simple digraph is strongly connected iff there is a cycle in G which includes each node at least once and no isolated node.

10 The *diameter* of a simple digraph $G = \langle V, E \rangle$ is given by δ, where

$$\delta = \max_{u, v \in V} d(u, v)$$

Find the diameter of the digraphs given in Figs. 5-1.13 and 5-1.14.

11 Find the strong components of the digraph given in Fig. 5-1.14. Also find its unilateral and weak components.

12 Show that every node and edge of a graph are contained in exactly one weak component.

5-1.3 Matrix Representation of Graphs

A diagrammatic representation of a graph has limited usefulness. Furthermore, such a representation is only possible when the number of nodes and edges is reasonably small. In this subsection we shall present an alternative method of representing graphs using matrices. Such a method of representation has several advantages. It is easy to store and manipulate matrices and hence the graphs represented by them in a computer. Well-known operations of matrix algebra can be used to calculate paths, cycles, and other characteristics of a graph.

Given a simple digraph $G = \langle V, E \rangle$, it is necessary to assume some kind of ordering of the nodes of the graph in the sense that a particular node is called a first node, another a second node, and so on. Our matrix representation of G depends upon the ordering of the nodes.

Definition 5-1.12 Let $G = \langle V, E \rangle$ be a simple digraph in which $V = \{v_1, v_2, \ldots, v_n\}$ and the nodes are assumed to be ordered from v_1 to v_n. An $n \times n$ matrix A whose elements a_{ij} are given by

$$a_{ij} = \begin{cases} 1 & \text{if } \langle v_i, v_j \rangle \in E \\ 0 & \text{otherwise} \end{cases}$$

is called the *adjacency matrix* of the graph G.

Recall that the adjacency matrix is the same as the relation matrix or the incidence matrix of the relation E in V. Any element of the adjacency matrix is either 0 or 1. Any matrix whose elements are either 0 or 1 is called a *bit matrix*

or a *Boolean matrix*. Note that the ith row in the adjacency matrix is determined by the edges which originate in the node v_i. The number of elements in the ith row whose value is 1 is equal to the outdegree of the node v_i. Similarly, the number of elements whose value is 1 in a column, say the jth column, is equal to the indegree of the node v_j. An adjacency matrix completely defines a simple digraph.

For a given digraph $G = \langle V, E \rangle$, an adjacency matrix depends upon the ordering of the elements of V. For different orderings of the elements of V we get different adjacency matrices of the same graph G. However, any one of the adjacency matrices of G can be obtained from another adjacency matrix of the same graph by interchanging some of the rows and the corresponding columns of the matrix. We shall neglect the arbitrariness introduced in an adjacency matrix because of the ordering of the elements of V and take any adjacency matrix of the graph to be the adjacency matrix of the graph. In fact, if two digraphs are such that the adjacency matrix of one can be obtained from the adjacency matrix of the other by interchanging some of the rows and the corresponding columns, then the digraphs are isomorphic.

As an example, consider the digraph given in Fig. 5-1.15 in which first we order the nodes as v_1, v_2, v_3, and v_4 and write its adjacency matrix. Next we reorder the rows as v_2, v_3, v_1, and v_4 and write its adjacency matrix. The two adjacency matrices are A_1 and A_2 as shown. If we interchange the first row and the first column with the third row and the third column of A_2, and then we interchange the second row with the third row and similarly the second column with the third column, we get A_1.

$$A_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad A_2 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Some of the properties of a simple digraph are immediately seen from its adjacency matrix. If a digraph is reflexive, then the diagonal elements of the adjacency matrix are 1s. For a symmetric digraph, the adjacency matrix is also symmetric; that is, $a_{ij} = a_{ji}$ for all i and j. Similarly, if a digraph is antisymmetric, then $a_{ij} = 1$ implies $a_{ji} = 0$, and $a_{ij} = 0$ implies that $a_{ji} = 1$ for all i and j.

We can extend the idea of matrix representation to multigraphs and weighted graphs. For simple undirected graphs, such an extension simply gives

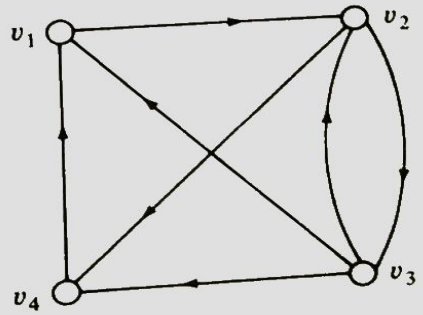


FIGURE 5-1.15

a symmetric adjacency matrix. In the case of a multigraph or a weighted graph, we write $a_{ij} = w_{ij}$ where w_{ij} denotes either the multiplicity or the weight of the edge $\langle v_i, v_j \rangle$. If $\langle v_i, v_j \rangle \notin E$, then we write $w_{ij} = 0$.

For a null graph which consists of only n nodes but no edges, the adjacency matrix has all its elements zero; i.e., the adjacency matrix is a null matrix. If there are loops at each node but no other edges in the graph, then the adjacency matrix is the identity or the unit matrix. If $G = \langle V, E \rangle$ is a simple digraph whose adjacency matrix is A, then the adjacency matrix of $\tilde{G}$, the converse of G, is the transpose of A, that is, A^T. For an undirected graph or for a symmetric graph, $A = A^T$.

We shall now consider the matrices AA^T, A^TA, and A^n for $n = 2, 3, 4, \ldots$. As an example, let us choose $A = A_1$, the first adjacency matrix of the digraph given in Fig. 5-1.15. Some of the matrices obtained from A are as follows:

$$A^T = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} \qquad AA^T = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 1 & 0 \\ 1 & 1 & 3 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

$$A^TA = \begin{pmatrix} 2 & 1 & 0 & 1 \\ 1 & 2 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 2 \end{pmatrix} \qquad A^2 = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 2 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$A^3 = \begin{pmatrix} 2 & 1 & 0 & 1 \\ 1 & 2 & 1 & 1 \\ 2 & 2 & 1 & 2 \\ 0 & 0 & 1 & 1 \end{pmatrix} \qquad A^4 = \begin{pmatrix} 1 & 2 & 1 & 1 \\ 2 & 2 & 2 & 3 \\ 3 & 3 & 2 & 3 \\ 2 & 1 & 0 & 1 \end{pmatrix}$$

Let us consider the elements of AA^T. For the sake of simplicity, write $B = AA^T$ and denote by b_{ij} the element in the ith row and jth column of B or AA^T. In general, for $i, j = 1, 2, \ldots, n$

$$b_{ij} = \sum_{k=1}^{n} a_{ik}a_{jk}$$

The value of $a_{ik}a_{jk} = 1$ iff both a_{ik} and a_{jk} are 1; otherwise $a_{ik}a_{jk} = 0$. Now $a_{ik} = 1$ if $\langle v_i, v_k \rangle \in E$, and $a_{jk} = 1$ if $\langle v_j, v_k \rangle \in E$. If both $\langle v_i, v_k \rangle$ and $\langle v_j, v_k \rangle$ are the edges of the graph for some fixed k, then we get a contribution of 1 in the summation expressing b_{ij}. The value of b_{ij} shows the number of nodes which are terminal nodes of edges from both v_i and v_j. In the graph G of Fig. 5-1.15, choose $i = 2$ and $j = 3$ and note that only the node v_4 is such that the edges from both

v_2 and v_3 terminate in v_4. Hence the entry is 1 in the second row and third column. If $i = j$, then

$$b_{ii} = \sum_{k=1}^{n} a_{ik}^2$$

and $a_{ik}^2 = 1$ if $a_{ik} = 1$, that is, if $\langle v_i, v_k \rangle \in E$. The diagonal entries of AA^T simply show the outdegree of the nodes.

A similar discussion shows that the element in the ith row and jth column of A^TA shows the number of nodes of the graph which are such that the edges initiating from these nodes terminate in both v_i and v_j. Also the diagonal entries show the indegrees of the nodes.

Consider now the powers of an adjacency matrix. Naturally an entry of 1 in the ith row and jth column of A shows the existence of an edge $\langle v_i, v_j \rangle$, that is, a path of length 1 from v_i to v_j. Let us denote the elements of A^2 by $a_{ij}^{(2)}$. Then

$$a_{ij}^{(2)} = \sum_{k=1}^{n} a_{ik}a_{kj}$$

For any fixed k, $a_{ik}a_{kj} = 1$ iff both a_{ik} and a_{kj} equal 1, that is, iff $\langle v_i, v_k \rangle$ and $\langle v_k, v_j \rangle$ are the edges of the graph. For each such k we get a contribution of 1 in the sum. Now $\langle v_i, v_k \rangle$ and $\langle v_k, v_j \rangle$ imply that there is a path from v_i to v_j of length 2. Therefore, $a_{ij}^{(2)}$ is equal to the number of different paths of exactly length 2 from v_i to v_j. Similarly, the diagonal element $a_{ii}^{(2)}$ shows the number of cycles of length 2 at the node v_i for $i = 1, 2, \ldots, n$.

By a similar argument, one can show that the element in the ith row and jth column of A^3 gives the number of paths of exactly length 3 from v_i to v_j. In general, we have the following theorem.

Theorem 5-1.3 Let A be the adjacency matrix of a digraph G. The element in the ith row and jth column of A^n (n is a nonnegative integer) is equal to the number of paths of length n from the ith node to the jth node.

Theorem 5-1.3 can be proved for any positive integer n by using mathematical induction and an argument similar to the one given here.

For the graph given in Fig. 5-1.15 we see that there are two paths of length 2 from v_2 to v_1, hence the entry 2 in the second row and first column of A^2. Similarly, there are two paths of length 4 from v_2 to v_1, hence the corresponding entry in A^4.

Given a simple digraph $G = \langle V, E \rangle$, let v_i and v_j be any two nodes of G. From the adjacency matrix of A we can immediately determine whether there exists an edge from v_i to v_j in G. Also from the matrix A^r, where r is some positive integer, we can establish the number of paths of length r from v_i to v_j. If we add the matrices $A, A^2, A^3, \ldots, A^r$ to get B_r

$$B_r = A + A^2 + \cdots + A^r$$

then from the matrix B_r we can determine the number of paths of length less than or equal to r from v_i to v_j. If we wish to determine whether v_j is reachable from v_i, it would be necessary to investigate whether there exists a path of any

length from v_i to v_j. In order to decide this, with the help of the adjacency matrix we would have to consider all possible A^r for $r = 1, 2, \ldots$. This method is neither practical nor necessary, as we shall show.

Recall that in a simple digraph with n nodes, the length of an elementary path or cycle does not exceed n (see Theorem 5-1.1). Also for a path between any two nodes one can obtain an elementary path by deleting certain parts of the path which are cycles. Similarly (for cycles), we can always obtain an elementary cycle from a given cycle. If we are interested in determining whether there exists a path from v_i to v_j, all we need to examine are the elementary paths of length less than or equal to $n - 1$. In the case of $v_i = v_j$ and the path is a cycle, we need to examine all possible elementary cycles of length less than or equal to n. Such cycles or paths are easily determined from the matrix B_n where

$$B_n = A + A^2 + A^3 + \cdots + A^n$$

The element in the ith row and jth column of B_n shows the number of paths of length n or less which exist from v_i to v_j. If this element is nonzero, then it is clear that v_j is reachable from v_i. Of course, in order to determine reachability, we need to know the existence of a path, and not the number of paths between any two nodes. In any case, the matrix B_n furnishes the required information about the reachability of any node of the graph from any other node.

Definition 5-1.13 Let $G = \langle V, E \rangle$ be a simple digraph in which $|V| = n$ and the nodes of G are assumed to be ordered. An $n \times n$ matrix P whose elements are given by

$$p_{ij} = \begin{cases} 1 & \text{if there exists a path from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

is called the *path matrix* (*reachability matrix*) of the graph G.

Note that the path matrix only shows the presence or absence of at least one path between a pair of points and also the presence or absence of a cycle at any node. It does not, however, show all the paths that may exist. In this sense a path matrix does not give complete information about a graph as does the adjacency matrix. The path matrix is important in its own right.

The path matrix can be calculated from the matrix B_n by choosing $p_{ij} = 1$ if the element in the ith row and jth column of B_n is nonzero and $p_{ij} = 0$ otherwise. We shall apply this method of calculating the path matrix to our sample problem, whose graph is given in Fig. 5-1.15. The adjacency matrix $A = A_1$ and its powers A^2, A^3, A^4 have already been calculated. We thus have B_4 and the path matrix P given by

$$B_4 = \begin{bmatrix} 3 & 4 & 2 & 3 \\ 5 & 5 & 4 & 6 \\ 7 & 7 & 4 & 7 \\ 3 & 2 & 1 & 2 \end{bmatrix} \qquad P = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

It may be remarked that if we are interested in knowing the reachability of one node from another, it is sufficient to calculate B_{n-1}, because a path of length n cannot be elementary. The only difference between P calculated from B_{n-1} and P calculated from B_n is in the diagonal elements. For the purpose of reachability, every node is assumed to be reachable from itself. Some authors calculate the path matrix from B_{n-1}, while others do it from B_n.

The method of calculating the path matrix P of a graph by calculating first A, A^2, . . ., A^n and then B_n is cumbersome. We shall now describe another method based upon a similar idea but which is more efficient in practice. Observe that we are not interested in the number of paths of any particular length from a node, say v_i, to a node v_j. This information is obtained during the course of our calculation of the powers of A, and later it is suppressed because these actual numbers are not needed. To reduce the amount of calculation involved, this unwanted information is not generated. This is achieved by using Boolean matrix operations in our calculations, which will now be defined.

A matrix whose entries are the elements of a two-element Boolean algebra $\langle B, \wedge, \vee, \bar{}, 0, 1 \rangle$ where $B = \{0, 1\}$ is called a Boolean matrix. The operations $\wedge$ and $\vee$ on B are given in Table 5-1.1. For any two $n \times n$ Boolean matrices A and B, the Boolean sum and Boolean product of A and B are written as $A \vee B$ and $A \wedge B$, which are also Boolean matrices, say C and D. The elements of C and D are given by

$$c_{ij} = a_{ij} \vee b_{ij} \qquad \text{and} \qquad d_{ij} = \bigvee_{k=1}^{n} (a_{ik} \wedge b_{kj}) \qquad \text{for all } i, j = 1, 2, \ldots, n$$

Note that the element d_{ij} is easily obtained by scanning the ith row of A from left to right and simultaneously the jth column of B from top to bottom. If, for any k, the kth element in the row and kth element in the column are both 1, then $d_{ij} = 1$; otherwise $d_{ij} = 0$.

The adjacency matrix is a Boolean matrix, and so also is the path matrix. Let us write $A \wedge A = A^{(2)}$, $A \wedge A^{(r-1)} = A^{(r)}$ for any $r = 2, 3, \ldots$. The only difference between A^2 and $A^{(2)}$ is that $A^{(2)}$ is a Boolean matrix and the entry in the ith row and jth column of $A^{(2)}$ is 1 if there is at least one path of length 2 from v_i to v_j, while in A^2 the entry in the ith row and jth column shows the number of paths of length 2 from v_i to v_j. Similar remarks apply to A^3 and $A^{(3)}$ or in general A^r and $A^{(r)}$ for any positive integer r. From this description, it is clear that the path matrix P is given by

$$P = A \vee A^{(2)} \vee A^{(3)} \vee \cdots \vee A^{(n)} = \bigvee_{k=1}^{n} A^{(k)}$$

If we take the sum from $k = 1$ to $k = n - 1$, we get a matrix which may differ if at all from P in the diagonal terms only.

Table 5-1.1

$\wedge$	0	1	$\vee$	0	1
0	0	0	0	0	1
1	0	1	1	1	1

For our sample example of the graph given in Fig. 5-1.15,

$$A^{(2)} = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \qquad A^{(3)} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \qquad A^{(4)} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

$$A \vee A^{(2)} \vee A^{(3)} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} = A \vee A^{(2)} \vee A^{(3)} \vee A^{(4)} = P$$

The matrices A, $A^{(2)}$, $A^{(3)}$, ... can be interpreted in a different way. In a simple digraph, $G = \langle V, E \rangle$, $E \subseteq V \times V$ so that E can be interpreted as a relation in V. The adjacency matrix A is the relation matrix of the relation E. Recall that the composite relation $E \circ E = E^2$ was defined in Sec. 2-3.7 as the relation such that $v_i \, E^2 \, v_j$ if there exists a v_k such that $v_i \, E \, v_k$ and $v_k \, E \, v_j$. In other words, the relation matrix of E^2 has 1 in the ith row and jth column if there is at least a path of length 2 from v_i to v_j. This shows that $A^{(2)}$ is the relation matrix of the relation E^2. Similarly, $A^{(3)}$, $A^{(4)}$, ... are the relation matrices of the relations $E \circ E \circ E = E^3, E^4, \ldots$ in V.

Next, let E_1 and E_2 be two relations in V and A_1 and A_2 be the corresponding relation matrices. For the relations $E_1 \cup E_2$ and $E_1 \cap E_2$, the relation matrices are given by $A_1 \vee A_2$ and $A_1 \wedge A_2$ respectively.

For a given relation E in V, a relation E^+, called the transitive closure of E, was defined in Sec. 2-3.7 as

$$E^+ = E \cup E^2 \cup \cdots$$

Clearly, the relation matrix of E^+ is given by

$$A^+ = A \vee A^{(2)} \vee A^{(3)} \vee \cdots$$

where A is the relation matrix of E. It has been shown that if the number of elements in V is n, then no path or cycle exceeds n in length; therefore A^+ can be obtained by simply considering the sum up to $A^{(n)}$, for powers higher than n will not change A^+. Therefore

$$A^+ = A \vee A^{(2)} \vee A^{(3)} \vee \cdots \vee A^{(n)} = P$$

The matrix A^+ is the same as the path matrix. This method of obtaining the transitive closure of a relation as well as the path matrix of a simple digraph can easily be programmed by using the following algorithm due to Warshall.

Let v_i be any node of a simple digraph G and P be its path matrix. If P' is the transpose of the matrix P, then the ith row of the matrix $P \wedge P'$ which is obtained by the elementwise product of the elements gives the strong component containing v_i.

Notice that if v_j is reachable from v_i, then clearly $p_{ij} = 1$; also, if v_i is reachable from v_j, then $p_{ji} = 1$ or $p'_{ij} = 1$. Therefore, the element in the ith row and jth column of $P \wedge P'$ is 1 iff v_i and v_j are mutually reachable. This is true for all j. Hence the result.

We shall end this subsection by showing how the path matrix of a digraph can be used in determining whether certain procedures in a program are recursive.

In some programming languages, a programmer must explicitly state that a procedure is recursive. For example, in PL/I the RECURSIVE option must be specified. In other languages which do not require any such specification, it is possible to use concepts from graph theory to determine which procedures are recursive. A recursive procedure is not necessarily one which invokes itself directly. If procedure p_1 invokes p_2, procedure p_2 invokes p_3, ..., procedure p_{n-1} invokes p_n, and procedure p_n invokes p_1, then procedure p_1 is recursive.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of procedures in a program. In a directed graph consisting of nodes representing elements of P, there is an edge from p_i to p_j if procedure p_i invokes p_j. Figure 5-1.16 shows a directed graph representing the calls made by the set of procedures $P = \{p_1, p_2, \ldots, p_5\}$. The adjacency matrix of the graph is

$$A = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{array} \begin{array}{c} \begin{array}{ccccc} p_1 & p_2 & p_3 & p_4 & p_5 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

A procedure p_i is recursive if there exists a cycle involving p_i in the graph. Such cycles can be detected from the diagonal elements of the path matrix

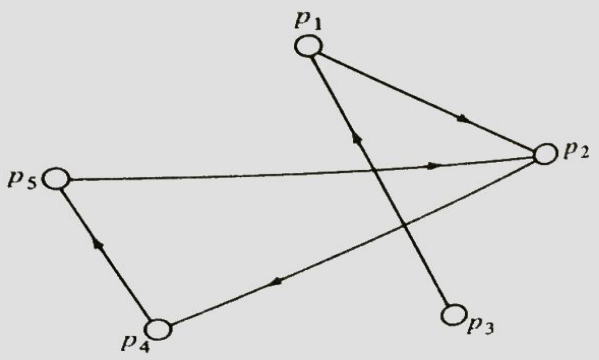


FIGURE 5-1.16 Procedure calls among p_1, p_2, p_3, p_4, and p_5.

$Q = A^+$ of the graph. Thus p_i is recursive iff $q_{ii} = 1$. The matrix Q can be obtained by using Warshall's algorithm. The matrix Q is given by

$$Q = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

which shows that the procedures p_2, p_4, and p_5 are recursive.

EXERCISES 5-1.3

1 Obtain the adjacency matrix A of the digraph given in Fig. 5-1.17. Find the elementary paths of lengths 1 and 2 from v_1 to v_4. Show that there is also a simple path of length 4 from v_1 to v_4. Verify the results by calculating A^2, A^3, and A^4.

2 For the digraph of Fig. 5-1.17 determine A', AA', and $A'A$. Interpret the entries of the matrix $A \wedge A'$. (A' is the transpose of A.)

3 For any $n \times n$ Boolean matrix A, show that

$$(I + A)^{(2)} = (I + A) \wedge (I + A) = I + A + A^{(2)}$$

where I is the $n \times n$ identity matrix and $A^{(2)} = A \wedge A$. Show also that for any positive integer r

$$(I + A)^{(r)} = I + A + A^{(2)} + \cdots + A^{(r)}$$

4 Using the result obtained in Prob. 3, show that the path matrix of a simple digraph is given by $P = (I + A)^{(n)}$ where A is the adjacency matrix of the digraph which has n nodes.

5 Given the adjacency matrix A of the digraph in Fig. 5-1.16, obtain the path matrix $Q = A^+$.

6 For a simple digraph $G = \langle V, E \rangle$ whose adjacency matrix is denoted by A, its *distance*

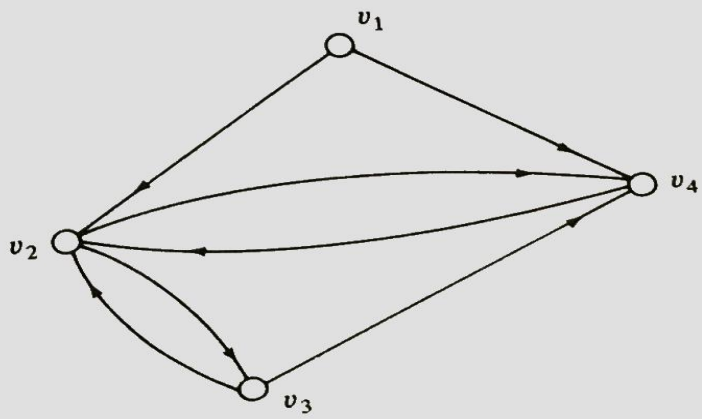


FIGURE 5-1.17

matrix is given by

$$d_{ij} = \infty \quad \text{if } \langle v_i, v_j \rangle \notin E$$
$$d_{ii} = 0 \quad \text{for all } i = 1, 2, \ldots, n$$
$$d_{ij} = k \quad \text{where } k \text{ is the smallest integer for which } a_{ij}^{(k)} \neq 0$$

Determine the distance matrix of the digraph given in Fig. 5-1.17. What does $d_{ij} = 1$ mean?

7 Show that a digraph G is strongly connected if all the entries of the distance matrix except the diagonal entries are nonzero. How will you obtain the path matrix from a distance matrix? How will you modify the diagonal entries?

8 Modify algorithm *MINIMA* so that all minimal paths are computed.

5-1.4 Trees

An important class of diagraphs called directed trees will be introduced in this section along with the terminology associated with such trees. Trees are useful in describing any structure which involves hierarchy. Familiar examples of such structures are family trees, the decimal classification of books in a library, the hierarchy of positions in an organization, an algebraic expression involving operations for which certain rules of precedence are prescribed, etc. We shall describe here how trees can be represented by diagrams and other means. Representation of trees in a computer is discussed in Sec. 5-2.1. Applications of trees to grammars is given in Sec. 5-3.1.

Definition 5-1.14 A *directed tree* is an acyclic digraph which has one node called its *root* with indegree 0, while all other nodes have indegree 1.

Note that every directed tree must have at least one node. An isolated node is also a directed tree.

Definition 5-1.15 In a directed tree, any node which has outdegree 0 is called a *terminal node* or a *leaf*; all other nodes are called *branch nodes*. The *level* of any node is the length of its path from the root.

The level of the root of a directed tree is 0, while the level of any node is equal to its distance from the root. Observe that all the paths in a directed tree are elementary, and the length of a path from any node to another node, if such a path exists, is the distance between the nodes, because a directed tree is acyclic.

Figure 5-1.18 shows three different diagrams of a directed tree. Several other diagrams of the same tree can be drawn by choosing different relative positions of the nodes with respect to its root. The directed tree of our example has two nodes at level 1, five nodes at level 2, and three nodes at level 3. Figure 5-1.18*a* shows a natural way of representation, viz., the way a tree grows from its root up and ending in leaves at different levels. Figure 5-1.18*b* shows the same tree drawn upside down. This is a convenient way of drawing a directed tree and is commonly used in the literature. Figure 5-1.18*c* differs from *b* in the order in which the nodes appear at any level from left to right. According to our defini-

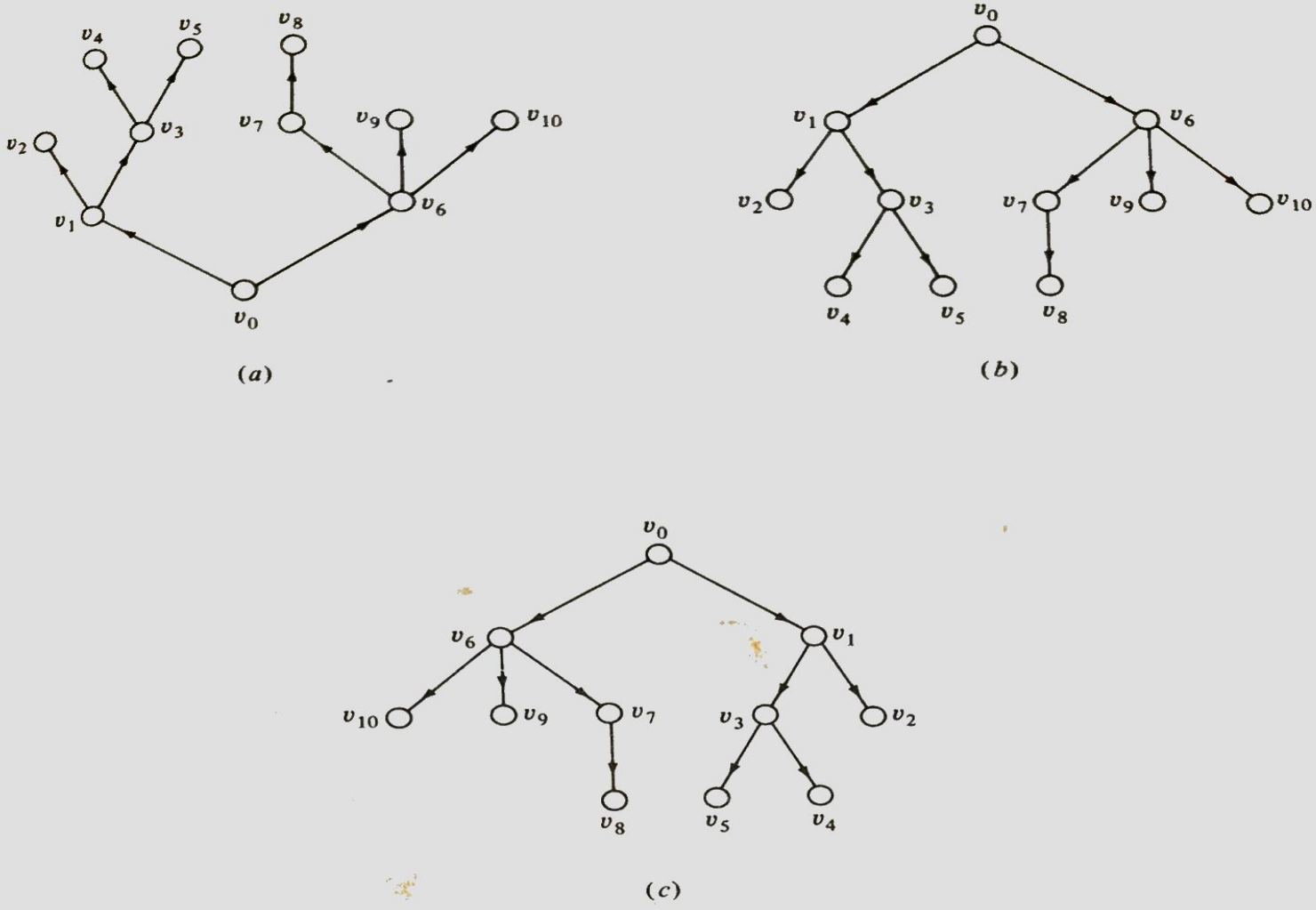


FIGURE 5-1.18

tion of a directed tree, such an order is of no significance. We shall, however, consider certain modifications so that an ordering of the nodes become relevant to a tree.

In many applications the relative order of the nodes at any particular level assumes some significance. In a computer representation such an order, even if it is arbitrary, is automatically implied. It is easy to impose an order on the nodes at a level by referring to a particular node as the first node, to another node as the second, and so on. In the diagrams the ordering may be done from left to right. Instead of ordering the nodes, we may prescribe an order on the edges. If, in a directed tree, an ordering of the nodes at each level is prescribed, then such a tree is called an *ordered tree*. According to this definition, the diagrams given in Fig. 5-18*b* and *c* represent the same directed tree but different ordered trees. Note that ordered trees as such are no longer directed graphs because the concept of order does not exist in a directed graph. We are mostly concerned with ordered trees in this section, and therefore we use the term "tree" to mean ordered tree unless stated otherwise. If we label the nodes as 1, 2, . . . or in some other way from left to right in an ordered tree, then such a tree is said to be *canonically labeled*.

In both directed and ordered trees it is important to decide whether the root is shown on top or at the bottom, because certain other terminology used to describe the relative positions of the nodes as above or below may assume different meanings according to the choice made for locating the root. In our discussion we shall assume that the root is at the top and that all other nodes are below the root.

From the structure of the directed tree it is clear that every node of a tree is the root of some subtrees contained in the original tree. In fact, if we delete the root and the edges connecting the root to the nodes at level 1, we get subtrees with roots which are the nodes at level 1. For the tree in Fig. 5-1.18, the node v_6 is the root of the subtree $\{v_6, v_7, v_8, v_9, v_{10}\}$, v_1 is the root of $\{v_1, v_2, v_3, v_4, v_5\}$, v_3 is the root of $\{v_3, v_4, v_5\}$, v_5 is the root of $\{v_5\}$, and v_7 is the root of $\{v_7, v_8\}$, etc. The number of subtrees of a node is called the *degree* of the node. Naturally, the degree of a terminal node is 0. The degree of v_3 is 2 because $\{v_4\}$ and $\{v_5\}$ are its subtrees, while the degree of v_1 is also 2 because $\{v_2\}$ and $\{v_3, v_4, v_5\}$ are its subtrees.

If we delete the root and the edges connecting the nodes at level 1, we obtain a set of disjoint trees. A set of disjoint trees is called a *forest*. We have also seen that any node of a directed tree is a root of some subtree. Therefore, subtrees immediately below a node form a forest.

At this stage we shall give a recursive definition of directed trees. According to this definition, a tree contains one or more nodes such that one of the nodes is called the root while all other nodes are partitioned into a finite number of trees called subtrees.

Here a tree with n nodes has been defined in terms of trees with less than n nodes. For the tree in Fig. 5-1.18, the tree $\{v_0, \ldots, v_{10}\}$ is defined in terms of trees $\{v_1, \ldots, v_5\}$ and $\{v_6, \ldots, v_{10}\}$, while the tree $\{v_1, \ldots, v_5\}$ can be defined in terms of $\{v_2\}$ and $\{v_3, v_4, v_5\}$, and so on. Finally, we get trees with one node each, which are its terminal nodes.

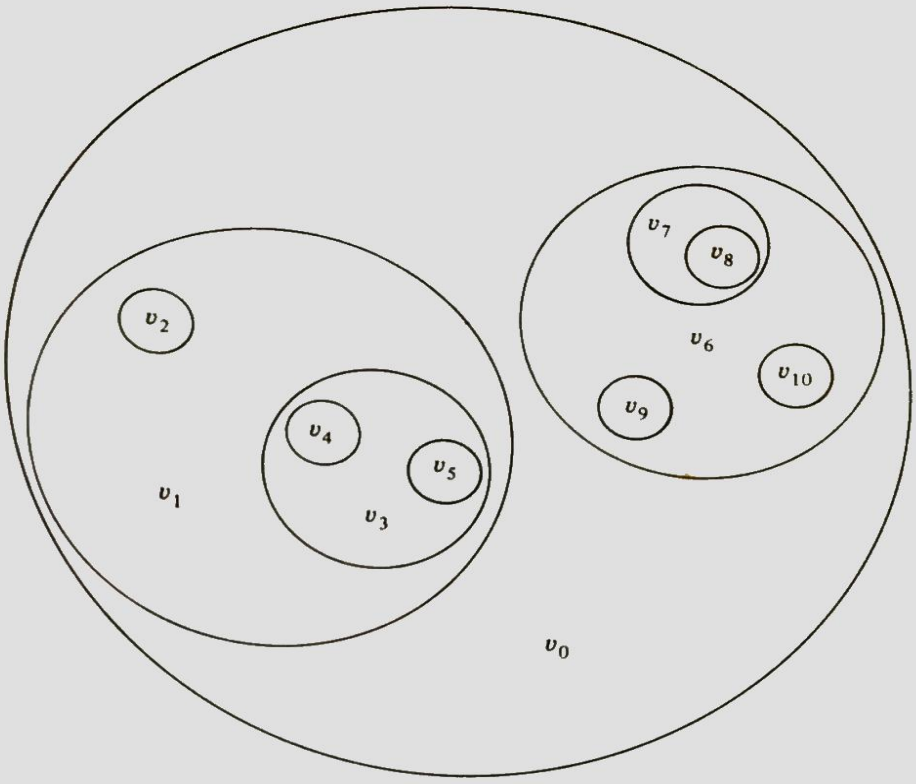
There are several other ways in which a directed tree can be represented graphically. These methods of representation for the directed tree of Fig. 5-1.18 are given in Fig. 5-1.19*a*, *b*, and *c*. The first method uses the familiar technique of Venn diagrams to show subtrees, the second uses the convention of nesting parentheses, and the last method is the one used in the list of contents of books.

The method of representation given in Fig. 5-1.19*b* immediately shows how any completely parenthesized algebraic expression or a well-formed formula in statement logic can be represented by a tree structure. Naturally, it is not necessary to have a completely parenthesized expression if we prescribe a set of precedence rules as discussed in Sec. 1-3.6. As an example, consider the expression

$$v_1 v_2 + \left(v_4 + \frac{v_5}{v_6}\right) v_3$$

The tree corresponding to this expression is shown in Fig. 5-1.20.

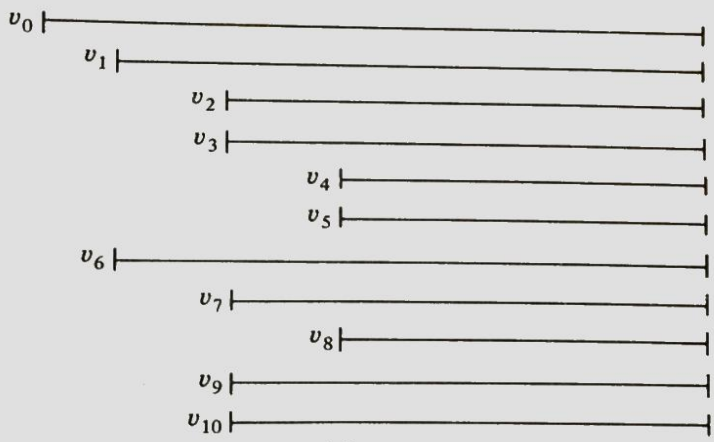
In the diagrams representing trees we have chosen to show the roots on top and the edges pointing downward. All the nodes at any particular level are shown on a horizontal line. In the case of an ordered tree, the nodes at any particular level are ordered from left to right. This ordering distinguishes an ordered tree from other directed trees. It is sometimes convenient to borrow some ter-



(*a*)

$(v_0(v_1(v_2)(v_3(v_4)(v_5)))(v_6(v_7(v_8))(v_9)(v_{10})))$

(*b*)



(*c*)

FIGURE 5-1.19 Different representations of trees.

minology from a family tree. Accordingly, every node that is reachable from a node, say u, is called a *descendent* of u. Also the nodes which are reachable from u through a single edge are called the *sons* of u.

So far we have not placed any restriction on the outdegree of any node in a directed or an ordered tree. If, in a directed tree, the outdegree of every node is less than or equal to m, then the tree is called an *m-ary tree*. If the outdegree

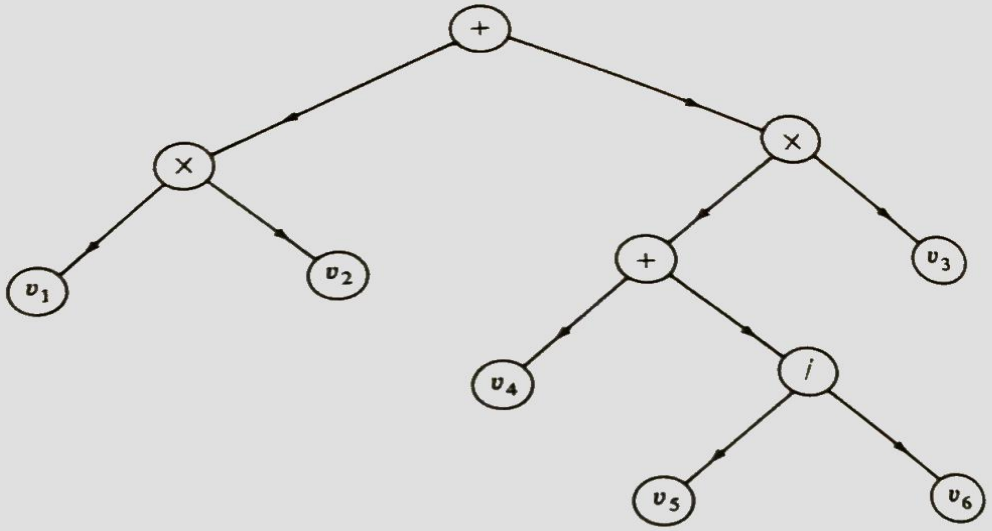
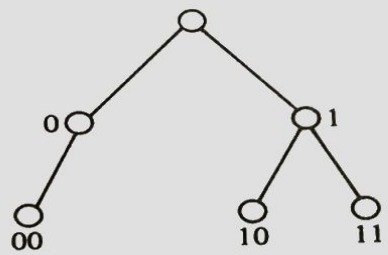


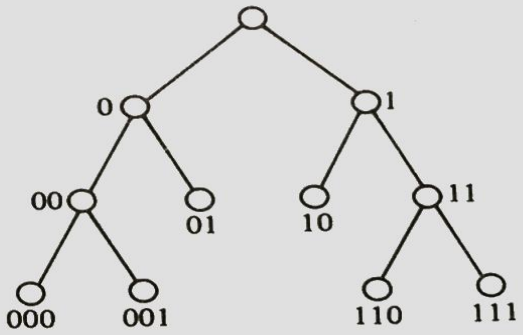
FIGURE 5-1.20

of every node is exactly equal to m or 0, then the tree is called a *full* or *complete* *m-ary tree*. For $m = 2$, the trees are called *binary* and *full binary* trees. We shall now consider m-ary trees in which the m (or fewer) sons of any node are assumed to have m distinct positions. If such positions are taken into account, then the tree is called a *positional m-ary tree.*

Figure 5-1.21*a* shows a binary tree, *b* shows a full binary tree, and *c* shows all four possible arrangements of sons of a node in a binary tree. The binary trees shown in Fig. 5-1.21*a* and *d* are distinct positional trees although they are not distinct ordered trees. In a positional binary tree, every node is uniquely

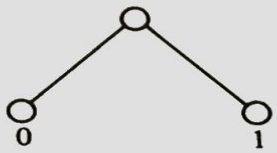


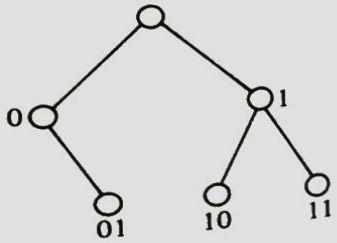
(a)



(b)

(c)





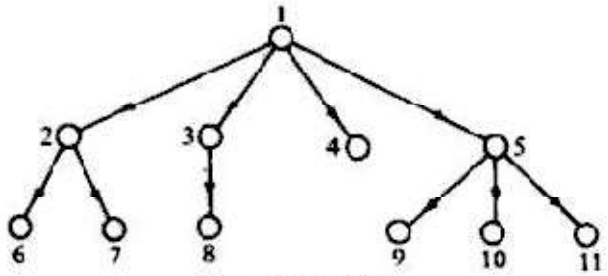
(d)

FIGURE 5-1.21

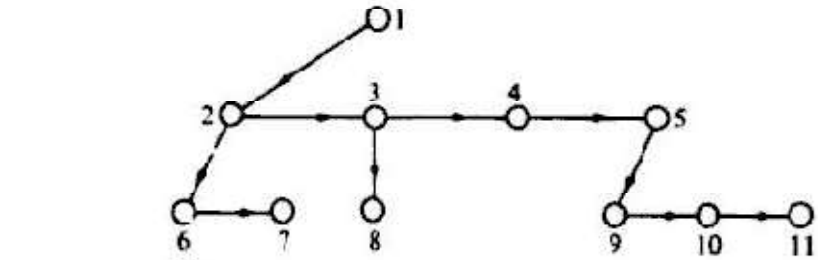
represented by a string over the alphabet {0, 1}, the root being represented by an empty string. Any son of a node u has a string which is prefixed by the string of u. The string of any terminal node is not prefixed to the string of any other node. The set of strings which correspond to terminal nodes form a *prefix* code. Thus the prefix code of the binary tree in Fig. 5-1.21*b* is {000, 001, 01, 10, 110, 111}. A similar representation of nodes of a positional m-ary tree by means of strings over an alphabet $\{0, 1, \ldots, m-1\}$ is possible.

The string representation of the nodes of a positional binary tree immediately suggests a natural method of representing a binary tree in a computer. It is sufficient for our purpose at this stage simply to recognize that such a natural representation exists.

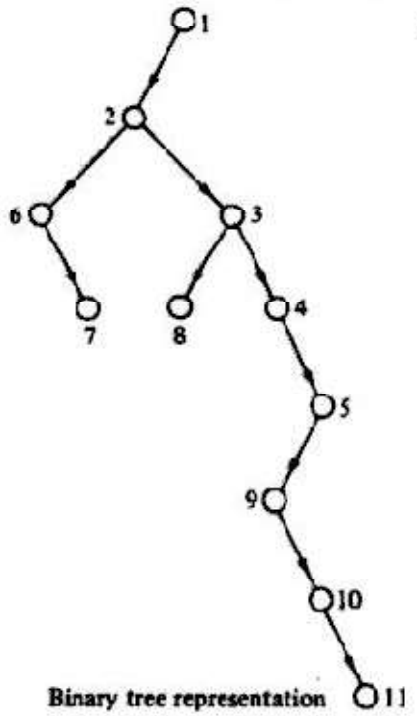
Binary trees are useful in several applications. We shall now show that every tree can be uniquely represented by a binary tree, so that for the computer representation of a tree it is possible to consider the representation of its



Given directed tree



Stage 1



Binary tree representation

corresponding binary tree. Furthermore a forest can also be represented by a binary tree.

In Fig. 5-1.22 we show in two stages how one can obtain a binary tree which represents a given ordered tree. As a first step, we delete all the branches originating in every node except the leftmost branch. Also, we draw edges from a node to the node on the right, if any, which is situated at the same level. Once this is done then for any particular node, we choose its left and right sons in the following manner. The left son is the node which is immediately below the given node, and the right son is the node to the immediate right of the given node on the same horizontal line. Such a binary tree will not have a right subtree.

The above method of representing any ordered tree by a unique binary tree can be extended to an ordered forest as shown in Fig. 5-1.23. Both these repre-

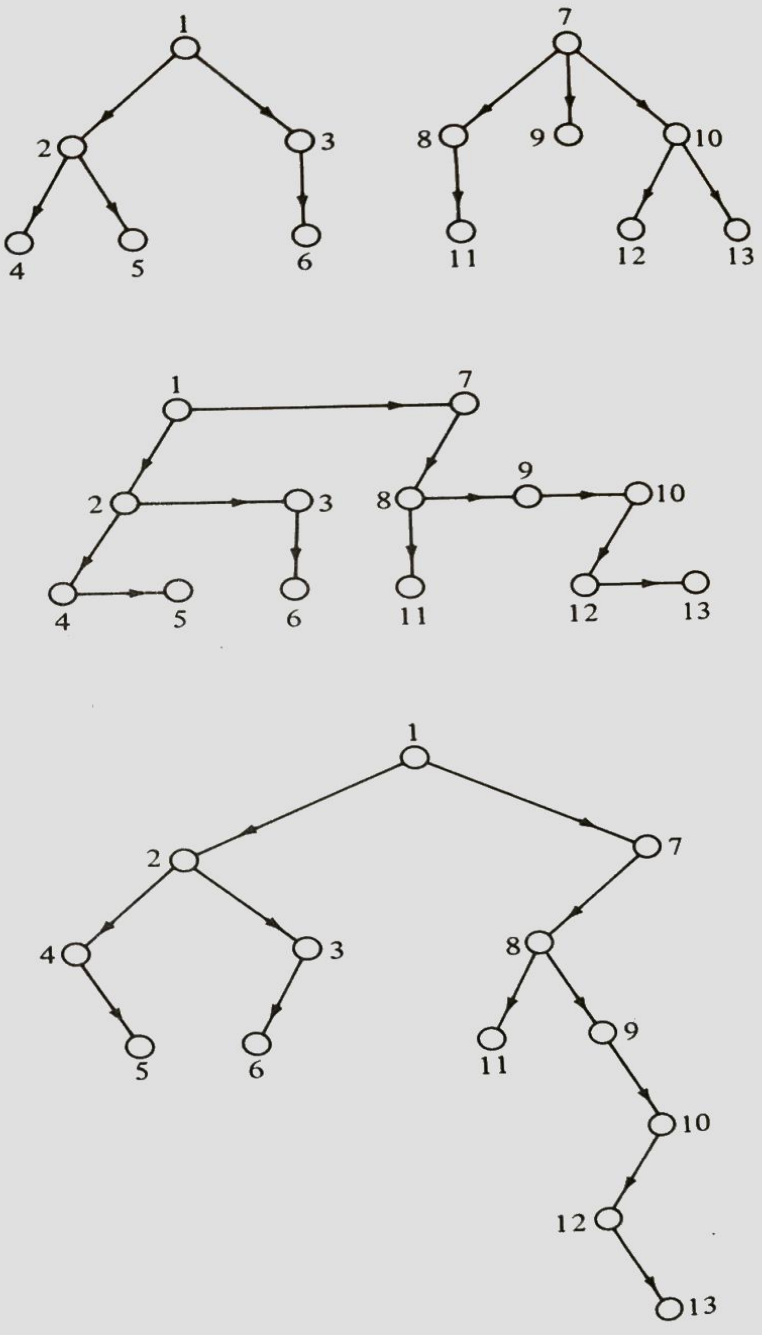


FIGURE 5-1.23 Binary tree representation of a forest.

sentations can be defined mathematically. This correspondence is called the natural correspondence between ordered trees and positional binary trees and also between ordered forests and positional binary trees.

EXERCISES 5-1.4

1 Show by means of an example that a simple digraph in which exactly one node has indegree 0 and every other node has indegree 1 is not necessarily a directed tree.
2 How many different directed trees are there with three nodes? How many different ordered trees are there with three nodes?
3 Give a directed tree representation of the following formula:

$$(P \vee (\neg P \wedge Q)) \wedge ((\neg P \vee Q) \wedge \neg R)$$

From this representation obtain the corresponding prefix formula.
4 Show that in a complete binary tree the total number of edges is given by $2(n_t - 1)$, where n_t is the number of terminal nodes.
5 From the adjacency matrix of a simple digraph, how will you determine whether it is a directed tree? If it is a directed tree, how will you determine its root and terminal nodes?
6 Obtain the binary tree corresponding to the tree given in Fig. 5-1.18.

5-2 STORAGE REPRESENTATION AND MANIPULATION OF GRAPHS

Recall that in Sec. 2-2 the computer representation of certain elementary discrete structures such as sets and arrays was discussed. We now wish to extend these concepts to more complex structures such as trees and graphs. Since trees are probably the most important nonlinear structure, their representations and manipulation will be emphasized in this section.

More particularly, our discussion will be limited to binary trees because their representation and manipulation are relatively simple when compared to those for general trees. Any general tree, as discussed at the end of Sec. 5-1.4, can be conveniently transformed into an equivalent binary tree.

The tree structures will be represented by using linked allocation. There are a number of storage methods that are based on sequential allocation, but we will not be concerned with them here.

The remainder of this section defines a list structure and describes a storage representation for it. These list structures are capable of representing certain digraphs. Finally, a storage method for representing a general graph is given.

5-2.1 Trees: Their Representation and Operations

The advantages and disadvantages in the use of linked allocation as opposed to sequential allocation in the representation of simple structures were discussed in Sec. 2-2. Although there are ways of representing trees based on sequential allocation techniques, we will not discuss them here. Computer representation of trees based on linked allocation seems to be more popular because of the ease

with which nodes can be inserted in and deleted from a tree, and because tree structures can grow to an arbitrary size, a size which is often unpredictable.

We will restrict our discussions to binary trees since they are easily represented and manipulated. A general tree can be readily converted into an equivalent binary tree by using the natural correspondence algorithm discussed in Sec. 5-1.4. Therefore linked allocation techniques will be used to represent binary trees. A number of possible traversals which can be performed on binary trees are described. The subsection ends with a symbol table algorithm based on a tree structure.

We now turn to the task of using linked allocation techniques to represent binary trees. Recall that a binary tree has one root node with no descendants or else a left, or a right, or a left and right subtree descendant(s). Each subtree descendant is also a binary tree, and we do make the distinction between its left and right branches. A convenient way of representing binary trees is to use linked allocation techniques involving nodes with structure

LLINK	*DATA*	*RLINK*

where *LLINK* or *RLINK* contain a pointer to the left subtree or right subtree respectively of the node in question. *DATA* contains the information which is to be associated with this particular node. Each pointer can have a value of *NULL*.

An example of a binary tree as a graph and its corresponding linked representation in memory are given in Fig. 5-2.1*a* and *b* respectively. Observe the very close similarity between the figures as drawn. Such a similarity illustrates that the linked storage representation of a tree is closer to the logical structuring of the data involved. This property can be useful in designing algorithms which process tree structures.

Let us now examine a number of operations which are performed on trees. One of the most common operations performed on tree structures is that of traversal. This is a procedure by which each node is processed exactly once in some systematic manner. Using the terminology popularized by Knuth, we can traverse a binary tree in three ways, namely, in preorder, in inorder, and in postorder. The following are recursive definitions for these traversals.

Preorder traversal

Process the root node.
Traverse the left subtree in preorder.
Traverse the right subtree in preorder.

Inorder traversal

Traverse the left subtree in inorder.
Process the root node.
Traverse the right subtree in inorder.

Postorder traversal

Traverse the left subtree in postorder.
Traverse the right subtree in postorder.
Process the root node.

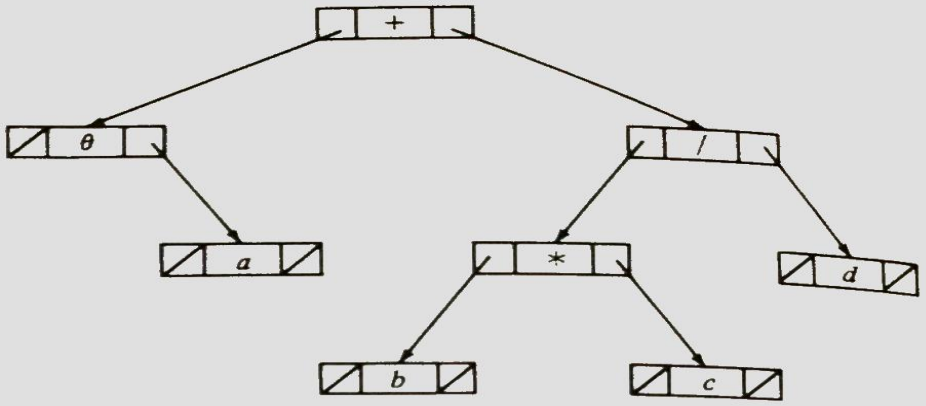


FIGURE 5-2.2 The formula $\ominus a + b * c/d$ as a binary tree.

and right subtrees are the left and right operands of that operator. The leaves of the tree will be the variables and constants in the expression. Let θ represent the unary minus. There are rules given in Exercises 3-4 that can distinguish a unary minus from a binary minus and the negative sign of a constant. The operand of θ will be considered to be a right subtree. The binary tree in Fig. 5-2.2 represents the formula $\theta a + b * c/d$. If we traverse this tree in preorder, we visit the nodes in the order $+\theta a/*bcd$, and this is merely the prefix form of the infix formula. On the other hand, if we traverse the tree in postorder, then we visit the nodes in the order $a\theta bc*d/+$, which is the formula written in suffix notation. Observe that if we had represented the formula as a general tree and then applied the natural correspondence algorithm of Sec. 5-1.4 to convert this tree into an equivalent binary tree, we would have the structures shown in Fig. 5-2.3. The prefix form of the formula is obtained by traversing the binary tree

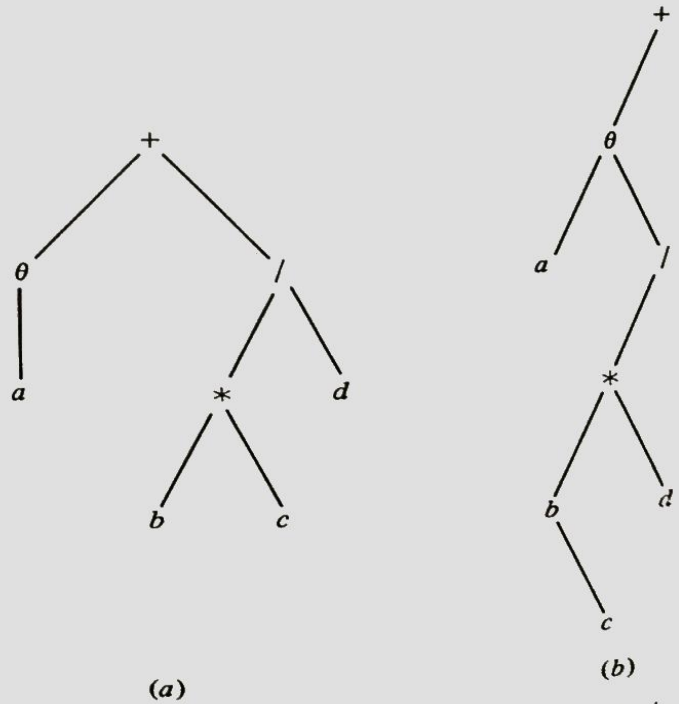


FIGURE 5-2.3 The general and binary tree representations of $\ominus a + b * c/d$. (*a*) General tree; (*b*) binary tree.

in preorder, while the suffix form is generated by an inorder traversal of the binary tree (not a postorder traversal!).

As an application of binary trees, we will formulate an algorithm that will maintain a tree-structured symbol table. (See Sec. 2-4.6.) One of the criteria that a symbol table routine must meet is that table searching be performed efficiently. This requirement originates in the compilation phase where many references to the entries of a symbol table are made. The two required operations that must be performed on a symbol table are insertion and "look-up," each of which involves searching. A binary tree structure was chosen for two reasons. The first reason is because if the symbol entries as encountered are randomly distributed according to lexicographic order, then table searching becomes approximately equivalent to a binary search as long as the tree is maintained in lexicographic order. Second, a binary tree structure is easily maintained in lexicographic order (in the sense that only a few pointers need be changed).

For simplicity, we assume a relatively sophisticated system which allows variable-length character strings to be used without much effort on the part of the programmer to handle them. We further assume that the symbol table routine is used to create trees that are local to a block of program code. This implies that an attempt to insert a duplicate entry is an error. In a global context, duplicate entries would be permitted as long as they were at different block levels. In a sense, the symbol table is a set of trees, one for each block level.

A binary tree will be constructed whose typical node is of the form

LLINK	*SYMBOLS*	*INFO*	*RLINK*

where *LLINK* and *RLINK* are pointer fields, *SYMBOLS* is the field for the character string which is the identifier or variable name (note that string descriptors might well be used here to allow fixed-length nodes, but it is assumed that this use is clear to the user), and *INFO* is some set of fields containing additional information about the identifier, such as its type. A node will be created by the execution of the statement $P \leftarrow NODE$ where the address of the new node is stored in P.

Finally, it is assumed that prior to any use of the symbol table routine at a particular block level, the appropriate tree head node is created with the *SYMBOLS* field set to a value that is greater lexicographically than any valid identifier. $HEAD[n]$ will point to this node where n designates the nth block level. Hence, the existence of an appropriate main routine which administers to the creation of tree heads as a new block is entered and to the deletion of tree heads as a block is exited, is assumed.

Because both the insertion and look-up operations involve many of the same actions (e.g., searching), we will actually produce only one routine, *TABLE*, and distinguish between insertion and look-up by the value of a global logical variable, *FLAG*. On invoking algorithm *TABLE*, if *FLAG* is *true*, then the requested operation is insertion; *NAME* and *DATA* contain the identifier name and additional information respectively. If the insertion is successful, then *FLAG* retains its original value; otherwise the value of *FLAG* is negated to indicate an error (because the identifier is already present in the table at that level), and an exit from the algorithm is made. On the other hand, if the algorithm is invoked with *FLAG* set to *false*, then the requested operation is look-up. In this

5-2.2 List Structures and Graphs

This subsection will first be concerned with the representation of a structure which is more general than a tree. Such a structure is called a list structure, and several programming languages have been developed to allow easy processing of structures similar to those that will be described. The need for list processing arose from the high cost of rapid computer storage and the unpredictable nature of the storage requirements of computer programs and data. There are many symbol manipulation applications in which this unpredictability is particularly acute. It will be shown that a list structure can be used to represent a directed graph. Second, we will give a brief introduction to the representation of a general graph structure. Such representations are based not only on the nature of the data, but also on the operations which are to be performed on the data. A specific representation for a particular application of graphs is given in Sec. 5-5.

In the context of list processing, we define a *list* to be any finite sequence of zero or more *atoms* or *lists*, where an atom is taken to be any object (e.g., a string of symbols) which is distinguishable from a list by treating the atom as structurally indivisible. If we enclose lists within parentheses and separate elements of lists by commas, then the following can be considered lists:

$(a, (b, c), d, (e, f, g))$

$( )$

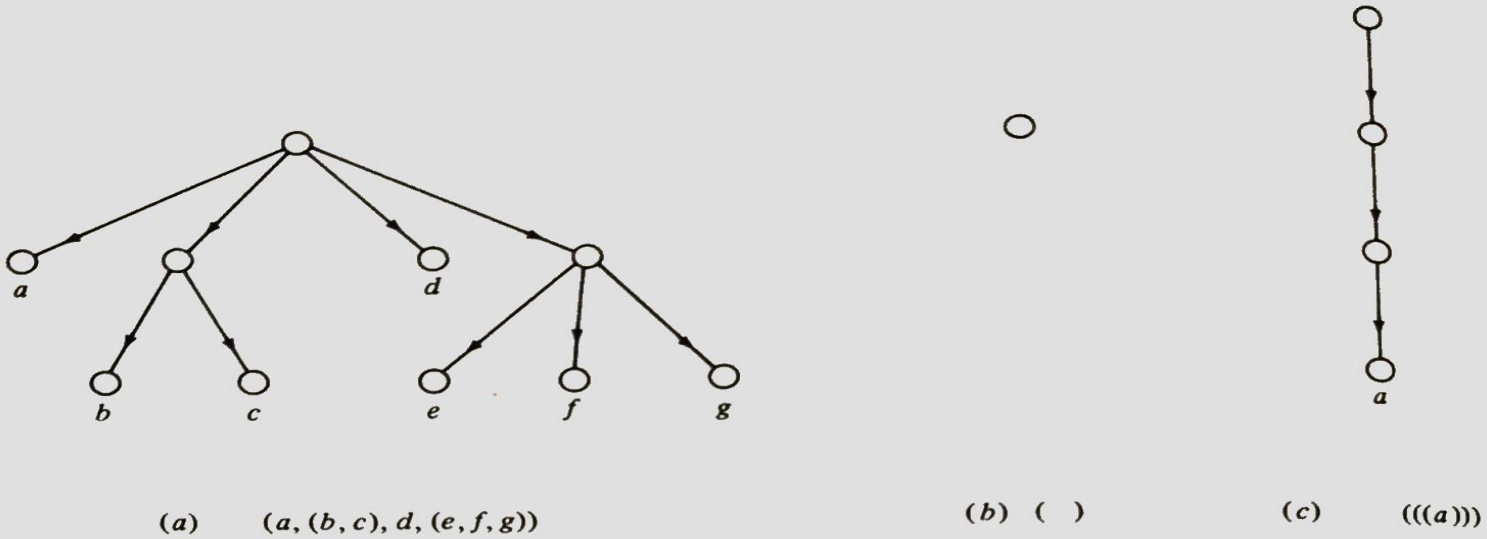
$(((a)))$

The first list contains four elements, namely, the atom a, the list (b, c) which contains the atoms b and c, the atom d, and the list (e, f, g) whose elements are the atoms e, f, and g. The second list has no elements, but the null list is still a valid list according to our definition. The third list has one element—the list $((a))$ which contains the single element (a), which in turn contains the atom a.

There is a distinct relationship between graphs and lists. A list is a directed graph with one *source* node (a node whose indegree is 0) corresponding to the entire list, and with every node immediately connected to the source node corresponding to an element of the list—either by being a node with outdegree 0 (for atoms) or by being a node that has branches (for elements which are lists) emanating from it. Every node except the source node has an indegree of 1. The edges leaving a node are considered to be ordered lists. This means that we distinguish the first edge, second edge, etc., which corresponds to the ordering of list elements by the first element, second element, etc. Furthermore, there are no cycles in the graph.

The preceding discussion could apply equally well to trees. However, lists are, in fact, extensions of trees in that a list can contain itself as an element and a tree cannot. Hence, there are some lists which cannot be represented as trees, but every tree can be represented as a list. Lists can have an essentially recursive nesting structure that no tree can have. Thus there are some lists that have a finite representation in our parentheses-comma notation, but which correspond to infinite graphs.

The graphs of some examples of lists appear in Fig. 5-2.4. If, however, M is the list (a, b, M), then we have an "infinite" graph for M, as shown in Fig. 5-2.5.



(a) (a, (b, c), d, (e, f, g)) (b) ( ) (c) (((a)))

FIGURE 5-2.4 Graph of list structures.

Linked allocation techniques can be used to represent lists in the memory of a computer. In such a representation there are two types of nodes—one for atoms and one for list elements. An atom node contains two fields: the first contains the value of the atom (e.g., a string of symbols), and the second contains a pointer to the next element in the list. A list node also contains two fields: the first field points to the storage representation of the list, and the second points to the element which follows this particular list element. It is assumed that an atom node and a list node are distinguishable.

Observe that in addition to order, a list also has *depth*. The list

$$(a, (b, c), d)$$

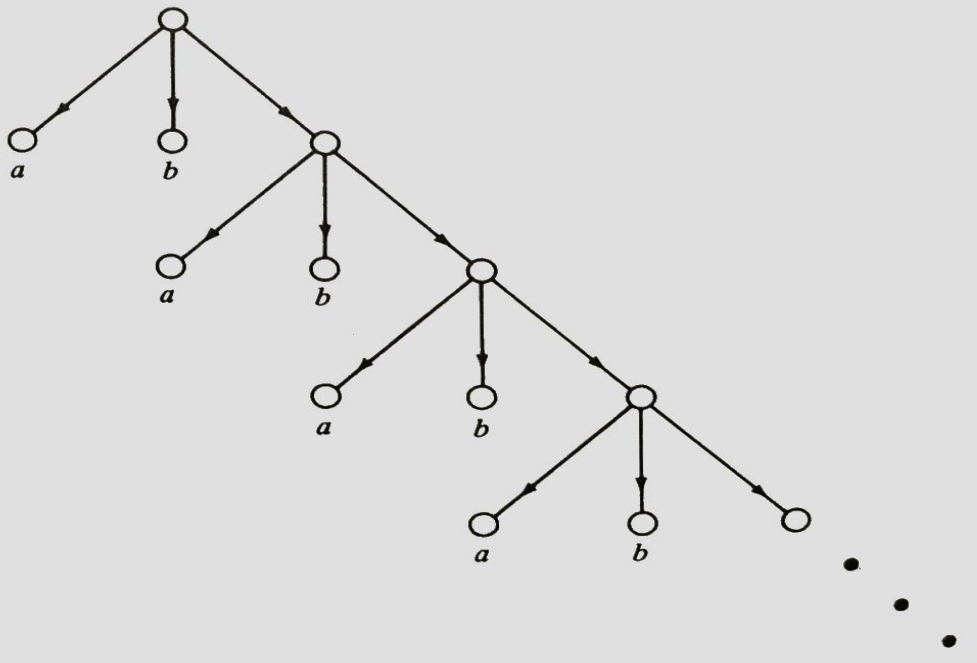


FIGURE 5-2.5 A recursive list structure and its graph.

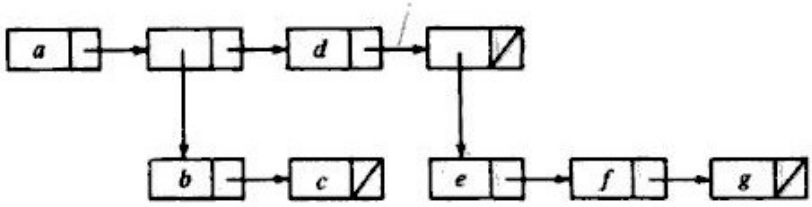


FIGURE 5-2.6 Storage representation of $(a, (b, c), d, (e, f, g))$.

has a depth of 2. The depth of a list is the number of levels it contains. The elements a and d are at level 1, and b and c are at level 2. The number of pairs of parentheses surrounding an element indicates its level. The element d in the list $(a, (b, (c, (d))))$ has a level of 4.

Order and depth are more easily understood in terms of our storage representation, where order is indicated by horizontal arrows and depth by vertical (downward-pointing) arrows. Thus, the list $(a, (b, c), d, (e, f, g))$ would be represented by the storage structure in Fig. 5-2.6. In storage, several lists may share common sublists. For example, the lists $(a, (b, c), d)$ and $(1, 5.2, (b, c))$ could be represented as in Fig. 5-2.7.

The recursive list M, where M is (a, b, M), can be represented as shown in Fig. 5-2.8. We would naturally use this storage representation where the common structure is shared rather than generating an infinite number of nodes to correspond to an infinite graph, but great care must be taken when manipulating such recursive structures in order to avoid programming oneself into an infinite loop.

A list structure occurs quite frequently in the processing of information, although it is not always evident. Consider a simple English sentence which consists of a subject, verb, and object. Any such sentence can be interpreted as a three-element list, whose elements can be atoms (single words) or lists (word phrases). The following sentences and their corresponding list representations are examples:

Man bites dog. = (Man, bites, dog)

The man bites the dog. = ((The, man), bites, (the, dog))

The big man is biting the small dog. = ((The, big, man), (is, biting), (the, small, dog))

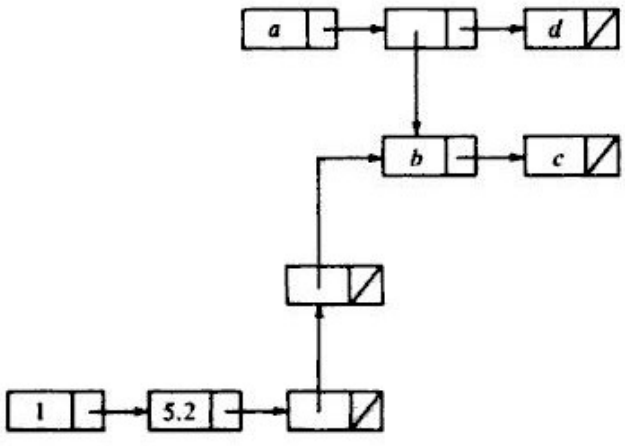


FIGURE 5-2.7 Two lists sharing a common sublist.

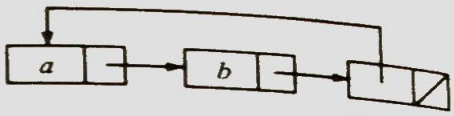


FIGURE 5-2.8 The representation of a recursive list M.

The subject and object of the last example can be further separated into nouns and qualifiers as in

(((The, big), (man)), (is, biting), ((the, small), (dog)))

The storage representation of this sentence is given in Fig. 5-2.9.

Observe that both order and level are significant. In this list structure, it is clear that the first node at level 1 either contains a noun which is the subject, or points to a list which contains a noun phrase which is the subject. If the latter is the case, we know that the first node (at level 2) of the list that is pointed to either contains a single qualifier or points to a list containing multiple qualifiers, and that the second node points to the noun. Thus, it is a simple matter to locate the noun subject of a sentence represented in this manner. Other elements of the sentence may be as easily isolated. Most list processing systems have features which facilitate such parsing operations.

List structures can therefore be used to represent digraphs, and a property of such representations is that sublists can be shared. As an example, a digraph and its list representation are given in Fig. 5-2.10.

As mentioned previously, several programming languages have been developed to allow easy processing of list structures. LISP 1.5 is one of the most powerful of these.

We will now discuss another storage method for graphs. The best storage representation for some general graph depends on the nature of the data and on the operations which are to be performed on these data. Furthermore, the choice of a suitable representation is affected by other factors such as the number of nodes, the average number of edges leaving a node, whether a graph is directed, the frequency of insertions and/or deletions to be performed, etc.

Arrays can sometimes be used (when there is at most one edge between any pair of nodes and there are no slings) to represent graphs. In this case the nodes are numbered from 1 to n, and a two-dimensional array with n rows and

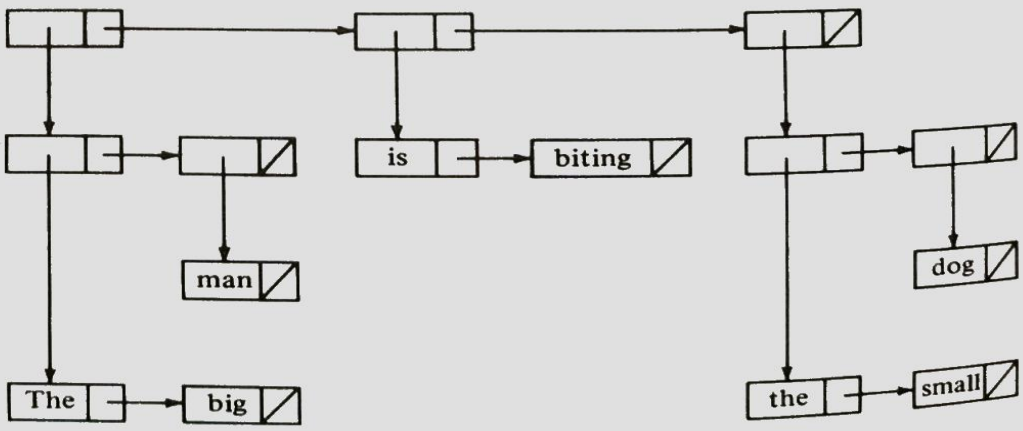


FIGURE 5-2.9 Storage representation of (((The, big), (man)), (is, biting), ((the, small), (dog))).

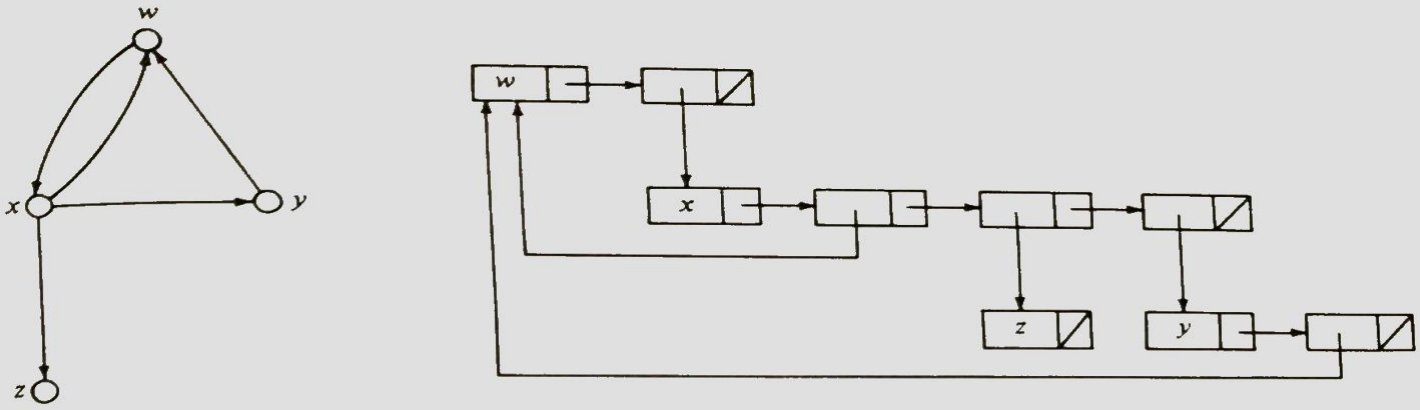
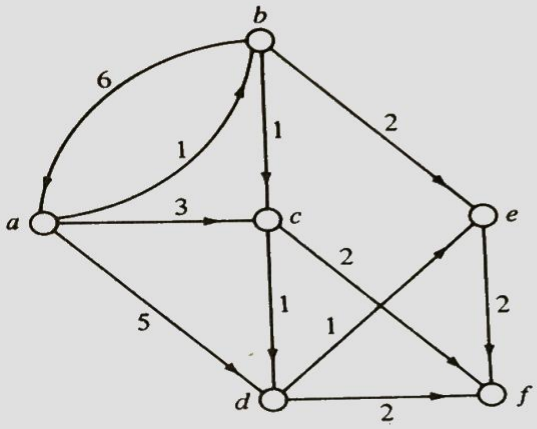


FIGURE 5-2.10 A digraph and its list structure representation.



Node #	Node data	Number of edges	Pointer to edge list
1	a	3	
2	b	3	
3	c	2	
4	d	2	
5	e	1	
6	f	0	NULL

Weight	Destination node
1	2
3	3
5	4

1	3
2	5
6	1

1	4
2	6

1	5
2	6

2	6

FIGURE 5-2.11 Storage representation of a weighted digraph.

n columns is used to represent the graph. Also, vectors could be required to store data on nodes in such a representation. This approach is not very suitable for a graph that has a large number of nodes or many nodes which are connected to only a few edges, nor when the graph must be continually altered.

If there are a number of branches between a pair of nodes and a considerable number of nodes that are connected to only a few other nodes, then a storage structure representation for such a graph could be the one shown in Fig. 5-2.11. Observe that the graph is weighted and that the storage representation consists of a node table directory, and associated with each entry in this directory we can have an edge list. A typical node directory entry consists of a node number, the data associated with it, the number of edges emanating from it, and a pointer field which gives the address of the edge list associated with this node. Each edge list, in this case, is stored as a sequential table whose typical entry consists of the weight of an edge and the node number at which that particular edge terminates. For a graph which is continually being changed, a representation which stores each edge list as a linked list is more desirable. In such a case it is not necessary to have the field which denotes the number of edges in the node table directory.

EXERCISES 5-2

1 Prove that a binary tree with n nodes has exactly $n + 1$ null branches.
2 Trace through algorithm *POSTORDER* using the binary tree of Fig. 5-2.1, and construct a table similar to Table 5-2.1.
3 Formulate an algorithm for the inorder traversal of a binary tree.
4 Trace through algorithm *TABLE* using a suitable set of seven variable names.
5 Given the binary tree in Fig. 5-2.12, determine the order in which the nodes will be visited if the tree is traversed in inorder, in postorder, and in preorder. Repeat this exercise for the converse traversals.

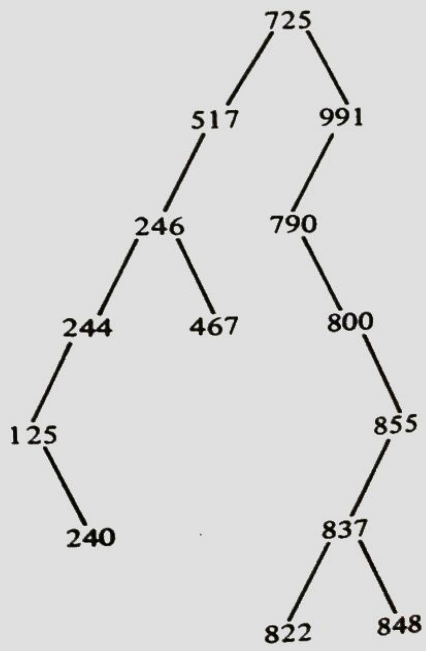


FIGURE 5-2.12

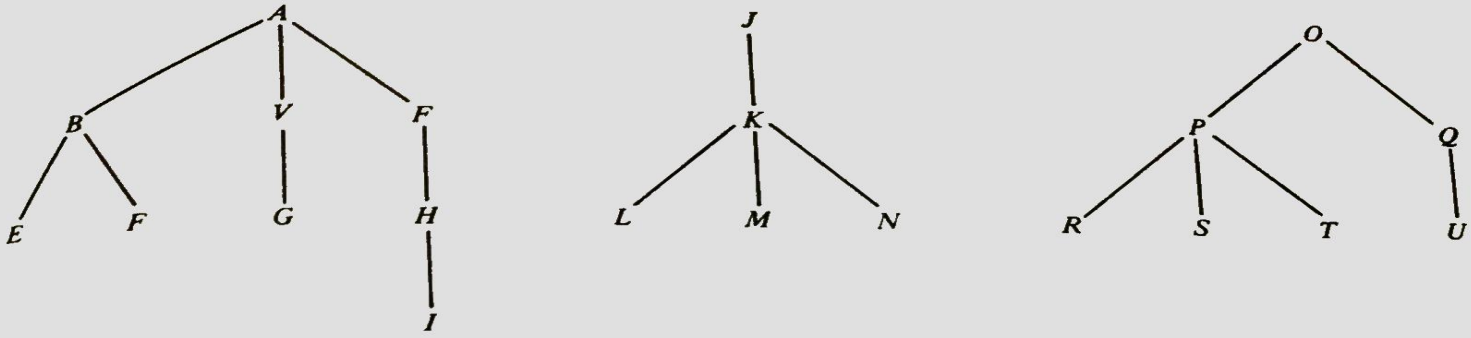


FIGURE 5-2.13 A forest of trees.

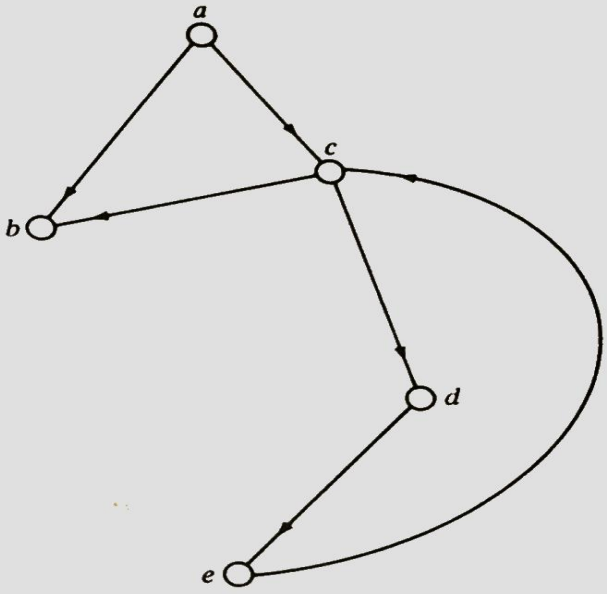


FIGURE 5-2.14

6 Develop an algorithm that converts a forest of trees into a single binary tree according to the natural correspondence. Illustrate the working of your algorithm by converting the forest in Fig. 5-2.13 to the binary tree corresponding to it.

7 Give a storage representation for the following lists:

$(a, (b, (c, d)), e, f)$

$((x), y, A, z)$ where $A = (a, b, (c, d))$

8 Represent the graph of Fig. 5-2.14 by a list structure. Draw its storage representation.