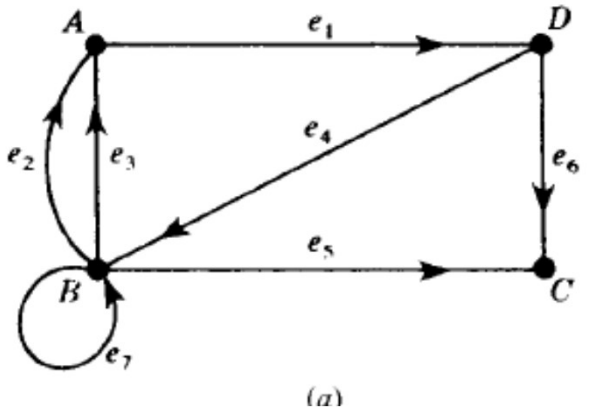
WEEK NINE

Basic Definitions

- A directed graph G or digraph (or simply graph) consists of two things:
- (i) A set V whose elements are called vertices, nodes, or points.
- (ii) A set E of ordered pairs (u,v) of vertices called arcs or directed edges or simply edges.
- Suppose e = (u,v) is a directed edge in a digraph G. Then the following terminology is used:
  - (a) e begins at u and ends at v.
  - (b) u is the origin or initial point of e, and v is the destination or terminal point of e.
  - (c) v is a successor of u.
  - (d) u is adjacent to v, and v is adjacent from u.

Basic Definitions



(a)

- If u = v, then e is called a loop.
- The set of all successors of a vertex u is important; it is denoted and formally defined by succ(u) = {v ∈ V|there exists an edge(u,v) ∈ E}
- It is called the successor list or adjacency list of u.
- Let G = G(V,E) be a directed graph, and let V ? be a subset of the set V of vertices of G. Suppose E' is a subset of E such that the endpoints of the edges in E' belong to V' . Then H(V',E') is a directed graph, and it is called a subgraph of G

Basic Definitions

- Degrees
- Suppose G is a directed graph. The outdegree of a vertex v of G, written outdeg(v), is the number of edges beginning at v, and the indegree of v, written indeg(v),is the number of edges ending at v. Since each edge begins and ends at a vertex we immediately obtain the following theorem.
- Theorem 9.1: The sum of the outdegrees of the vertices of a digraph G equals the sum of the indegrees of the vertices, which equals the number of edges in G.

Basic Definitions

- Let G be a directed graph. The concepts of path, simple path, trail, and cycle carry over from nondirected graphs to the directed graph G except that the directions of the edges must agree with the direction of the path.

(i) A (*directed*) *path* P in G is an alternating sequence of vertices and directed edges, say,

$$P = (v_0,\ e_1,\ v_1,\ e_2,\ v_2, \ldots, e_n,\ v_n)$$

such that each edge e_i begins at v_{i-1} and ends at v_i. If there is no ambiguity, we denote P by its sequence of vertices or its sequence of edges.

(ii) The *length* of the path P is n, its number of edges.

(iii) A *simple path* is a path with distinct vertices. A *trail* is a path with distinct edges.

(iv) A *closed path* has the same first and last vertices.

(v) A *spanning path* contains all the vertices of G.

(vi) A *cycle* (or *circuit*) is a closed path with distinct vertices (except the first and last).

(vii) A *semipath* is the same as a path except the edge e_i may begin at v_{i-1} or v_i and end at the other vertex. *Semitrails* and *semisimple paths* are analogously defined.

Basic Definitions

Connectivity

There are three types of connectivity in a directed graph G:

(i) G is *strongly connected* or *strong* if, for any pair of vertices u and v in G, there is a path from u to v and a path from v to u, that is, each is reachable from the other.

(ii) G is *unilaterally connected* or *unilateral* if, for any pair of vertices u and v in G, there is a path from u to v or a path from v to u, that is, one of them is reachable from the other.

(iii) G is *weakly connected* or *weak* if there is a semipath between any pair of vertices u and v in G.

Let G' be the (nondirected) graph obtained from a directed graph G by allowing all edges in G to be nondirected. Clearly, G is weakly connected if and only if the graph G' is connected.

Observe that strongly connected implies unilaterally connected which implies weakly connected. We say that G is *strictly unilateral* if it is unilateral but not strong, and we say that G is *strictly weak* if it is weak but not unilateral.

Connectivity can be characterized in terms of spanning paths as follows:

Theorem 9.2: Let G be a finite directed graph. Then:

(i) G is strong if and only if G has a closed spanning path.

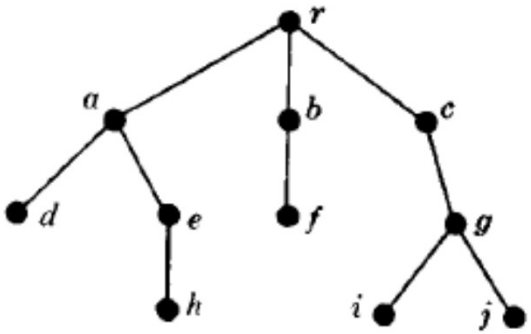
(ii) G is unilateral if and only if G has a spanning path.

(iii) G is weak if and only if G has a spanning semipath.

Rooted Trees

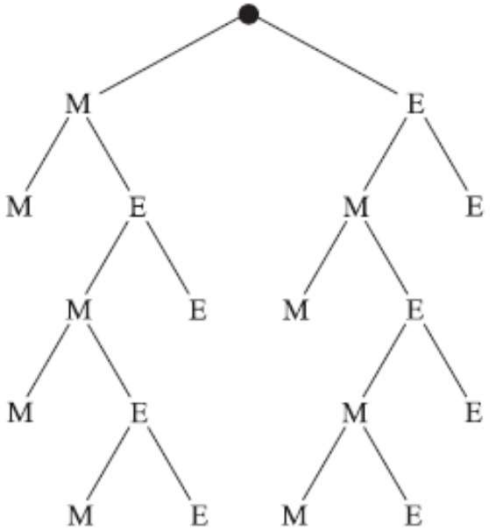
Recall that a tree graph is a connected cycle-free graph, that is, a connected graph without any cycles. A *rooted tree* T is a tree graph with a designated vertex r called the *root* of the tree. Since there is a unique simple path from the root r to any other vertex v in T, this determines a direction to the edges of T. Thus T may be viewed as a directed graph. We note that any tree may be made into a rooted tree by simply selecting one of the vertices as the root.

Consider a rooted tree T with root r. The length of the path from the root r to any vertex v is called the *level* (or *depth*) of v, and the maximum vertex level is called the *depth* of the tree. Those vertices with degree 1, other than the root r, are called the *leaves* of T, and a directed path from a vertex to a leaf is called a *branch*.



Rooted Trees

- Suppose Marc and Erik are playing a tennis tournament such that the first person to win two games in a row or who wins a total of three games wins the tournament. Find the number of ways the tournament can proceed.

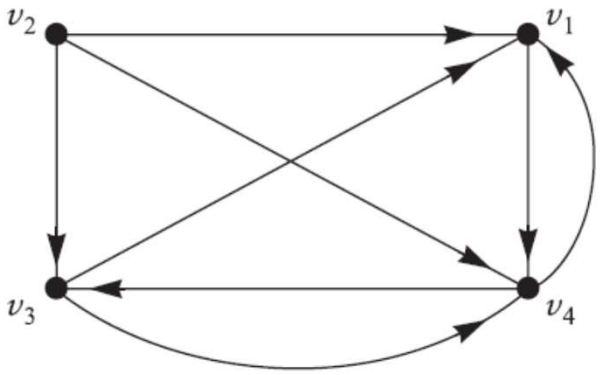


SEQUENTIAL REPRESENTATION OF DIRECTED GRAPHS

Suppose G is a simple directed graph with m vertices, and suppose the vertices of G have been ordered and are called $v_1, v_2, \ldots, v_m$. Then the *adjacency matrix* $A = [a_{ij}]$ of G is the $m \times m$ matrix defined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge } (v_i, v_j) \\ 0 & \text{otherwise} \end{cases}$$

Such a matrix A, which contains entries of only 0 or 1, is called a *bit matrix* or a *Boolean matrix*. (Although the adjacency matrix of an undirected graph is symmetric, this is not true here for a directed graph.)



(*a*)

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

(*b*)

SEQUENTIAL REPRESENTATION OF DIRECTED GRAPHS

Consider the powers A, A^2, A^3,... of the adjacency matrix $A = [a_{ij}]$ of a graph G. Let

$$a_K(i, j) = \text{the } ij \text{ entry in the matrix } A^K$$

Note that $a_1(i, j) = a_{ij}$ gives the number of paths of length 1 from vertex v_i to vertex v_j. One can show that $a_2(i, j)$ gives the number of paths of length 2 from v_i to v_j. In fact, we prove in Problem 9.17 the following general result.

Proposition 9.4: Let A be the adjacency matrix of a graph G. Then $a_K(i, j)$, the ij entry in the matrix A^K, gives the number of paths of length K from v_i to v_j.

SEQUENTIAL REPRESENTATION OF DIRECTED GRAPHS

EXAMPLE 9.7 Consider again the graph G and its adjacency matrix A appearing in Fig. 9-4. The powers A^2, A^3, and A^4 of A follow:

$$A^2 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 2 & 0 & 1 & 2 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 2 \end{bmatrix}, \quad A^3 = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 3 & 0 & 2 & 3 \\ 2 & 0 & 1 & 2 \\ 2 & 0 & 2 & 1 \end{bmatrix}, \quad A^4 = \begin{bmatrix} 2 & 0 & 2 & 1 \\ 5 & 0 & 3 & 5 \\ 3 & 0 & 2 & 3 \\ 3 & 0 & 1 & 4 \end{bmatrix}$$

Observe that $a_2(4, 1) = 1$, so there is a path of length 2 from v_4 to v_1. Also, $a_3(2, 3) = 2$, so there are two paths of length 3 from v_2 to v_3; and $a_4(2, 4) = 5$, so there are five paths of length 4 from v_2 to v_4.

Remark: Let A be the adjacency matrix of a graph G, and let B_r be the matrix defined by:

$$B_r = A + A^2 + A^3 + \cdots + A^r$$

Then the ij entry of the matrix B_r gives the number of paths of length r or less from vertex v_i to vertex v_j.

PATH MATRIX

Path Matrix

Let $G = G(V, E)$ be a simple directed graph with m vertices $v_1, v_2, \ldots, v_m$. The *path matrix* or *reachability matrix* of G is the m-square matrix $P = [p_{ij}]$ defined as follows:

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

Suppose now that there is a path from vertex v_i to vertex v_j in a graph G with m vertices. Then there must be a simple path from v_i to v_j when $v_i \neq v_j$, or there must be a cycle from v_i to v_j when $v_i = v_j$. Since G has m vertices, such a simple path must have length $m - 1$ or less, or such a cycle must have length m or less. This means that there is a nonzero ij entry in the matrix B_m (defined above) where A is the adjacency matrix of G. Accordingly, the path matrix P and B_m have the same nonzero entries. We state this result formally.

Proposition 9.5: Let A be the adjacency matrix of a graph G with m vertices. Then the path matrix P and B_m have the same nonzero entries where

$$B_m = A + A^2 + A^3 + \cdots + A^m$$

PATH MATRIX

Recall that a directed graph G is said to be *strongly connected* if, for any pair of vertices u and v in G, there is a path from u to v and from v to u. Accordingly, G is strongly connected if and only if the path matrix P of G has no zero entries. This fact together with Proposition 9.5 gives the following result.

Proposition 9.6: Let A be the adjacency matrix of a graph G with m vertices. Then G is strongly connected if and only if B_m has no zero entries where

$$B_m = A + A^2 + A^3 + \cdots + A^m$$

EXAMPLE 9.8 Consider the graph G and its adjacency matrix A appearing in Fig. 9-4. Here G has $m = 4$ vertices. Adding the matrix A and matrices A^2, A^3, A^4 in Example 9.7, we obtain the following matrix B_4 and also path (reachability) matrix P by replacing the nonzero entries in B_4 by 1:

$$B_4 = \begin{bmatrix} 4 & 0 & 3 & 4 \\ 11 & 0 & 7 & 11 \\ 7 & 0 & 4 & 7 \\ 7 & 0 & 4 & 7 \end{bmatrix} \quad \text{and} \quad P = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

Examining the matrix B_4 or P, we see zero entries; hence G is not strongly connected. In particular, we see that the vertex v_2 is not reachable from any of the other vertices.

Warshall's Algorith

Warshall's Algorithm

First we define m-square Boolean matrices $P_0, P_1, \ldots, P_m$ where $P_k[i, j]$ denotes the ij entry of the matrix P_k:

$$P_k[i, j] = \begin{cases} 1 & \text{if there is a simple path from } v_i \text{ to } v_j \text{ which does not use any} \\ & \quad \text{other vertices except possibly } v_1, v_2, \ldots, v_k, \\ 0 & \text{otherwise.} \end{cases}$$

For example,

$$P_3[i, j] = 1 \quad \text{if there is a simple path from } v_i \text{ to } v_j \text{ which does not use any other vertices except possibly } v_1, v_2, v_3.$$

Observe that the first matrix $P_0 = A$, the adjacency matrix of G. Furthermore, since G has only m vertices, the last matrix $P_m = P$, the path matrix of G.

Warshall observed that $P_k[i, j] = 1$ can occur only if one of the following two cases occurs:

Warshall's Algorithm

(1) There is a simple path from v_i to v_j which does not use any other vertices except possibly $v_1, v_2,\ldots,v_{k-1}$; hence

$$P_{k-1}[i, j] = 1$$

(2) There is a simple path from v_i to v_k and a simple path from v_k to v_j where each simple path does not use any other vertices except possibly $v_1, v_2,\ldots, v_{k-1}$; hence

$$P_{k-1}[i, k] = 1 \quad \text{and} \quad P_{k-1}[k, j] = 1$$

These two cases are pictured as follows:

$$(1)\ v_i \rightarrow \cdots \rightarrow v_j; \qquad (2)\ v_i \rightarrow \cdots \rightarrow v_k \rightarrow \cdots v_j$$

where $\rightarrow \cdots \rightarrow$ denotes part of a simple path which does not use any other vertices except possibly $v_1,v_2,\ldots,$ v_{k-1}. Accordingly, the elements of P_k can be obtained by:

$$P_k[i, j] = P_{k-1}[i, j] \ \vee \ (P_{k-1}[i, k] \ \wedge \ P_{k-1}[k, j])$$

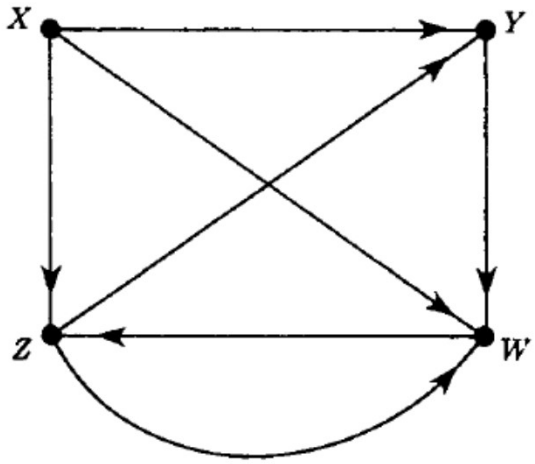
where we use the logical operations of a $\wedge$ (AND) and $\vee$ (OR). In other words we can obtain each entry in the matrix P_k by looking at only three entries in the matrix P_{k-1}. Warshall's algorithm appears in Fig. 9-6.

Boolean “and”/Boolean “or”

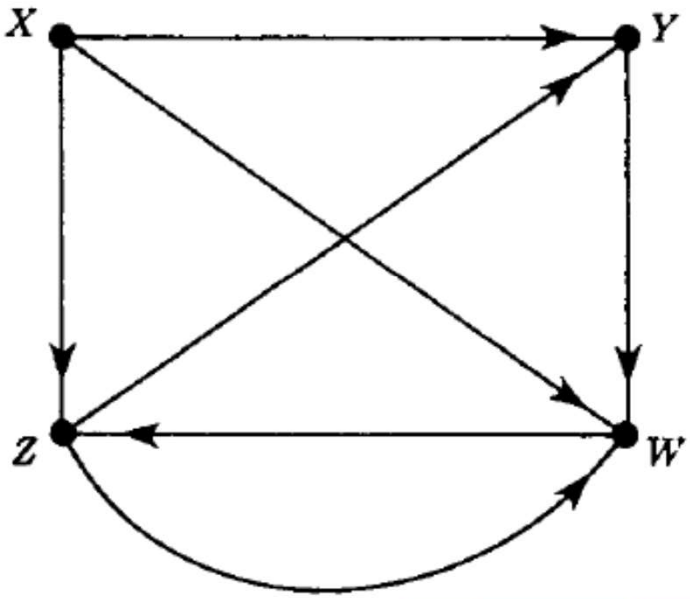
- AND
  - 1 ^ 1 = 1
  - 1 ^ 0 = 0
  - 0 ^ 1 = 0
  - 0 ^ 0 = 0
- OR
  - 1 v 1 = 1
  - 1 v 0 = 1
  - 0 v 1 = 1
  - 0 v 0 = 0

WARSHALL'S ALGORITHM

- Consider the adjacency matrix A of the following graph G obtained. Find the path matrix P of G using Warshall's algorithm
- Solution:



Warshall's Algorithm



Initially set $P_0 = A$. Then, P_1, P_2, P_3, P_4 are obtained recursively by setting

$$P_k[i, j] = P_{k-1}[i, j] \vee (P_{k-1}[i, k] \wedge P_{k-1}[k, j])$$

where P_k[i, j] denotes the ij-entry in the matrix P_k. That is, by setting

$$P_k[i, j] = 1 \quad \text{if} \quad P_{k-1}[i, j] = 1 \quad \text{or if both} \quad P_{k-1}[i, k] = 1 \quad \text{and} \quad P_{k-1}[k, j] = 1$$

Then matrices P_1, P_2, P_3, P_4 follow:

$$P_1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad P_2 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad P_3 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}, \quad P_4 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Observe that $P_1 = P_2 = A$. The changes in P_3 occur for the following reasons:

$$P_3[4, 2] = 1 \quad \text{because} \quad P_2[4, 3] = 1 \text{ and } P_2[3, 2] = 1$$
$$P_3[4, 4] = 1 \quad \text{because} \quad P_2[4, 3] = 1 \text{ and } P_2[3, 4] = 1$$

SHORTEST PATH ALGORITHM

Shortest-path Algorithm

Let G be a simple directed graph with m vertices, $v_1, v_2, \ldots, v_m$. Suppose G is weighted; that is, suppose each edge e of G is assigned a nonnegative number $w(e)$ called the *weight* or *length* of e. Then G may be maintained in memory by its weight matrix $W = [w_{ij}]$ defined as follows:

$$w_{ij} = \begin{cases} w(e) & \text{if there is an edge } e \text{ from } v_i \text{ to } v_j \\ 0 & \text{if there is no edge from } v_i \text{ to } v_j \end{cases}$$

The path matrix P tells us whether or not there are paths between the vertices. Now we want to find a matrix Q which tells us the lengths of the shortest paths between the vertices or, more exactly, a matrix $Q = [q_{ij}]$ where

$$q_{ij} = \text{length of the shortest path from } v_i \text{ to } v_j$$

Next we describe a modification of Warshall's algorithm which efficiently finds us the matrix Q.

Here we define a sequence of matrices $Q_0, Q_1, \ldots, Q_m$ (analogous to the above matrices $P_0, P_1, \ldots, P_m$) where $Q_k[i, j]$, the ij entry of Q_k, is defined as follows:

$Q_k[i, j]$ = the smaller of the length of the preceding path from v_i to v_j or the sum of the lengths of the preceding paths from v_i to v_k and from v_k to v_j.

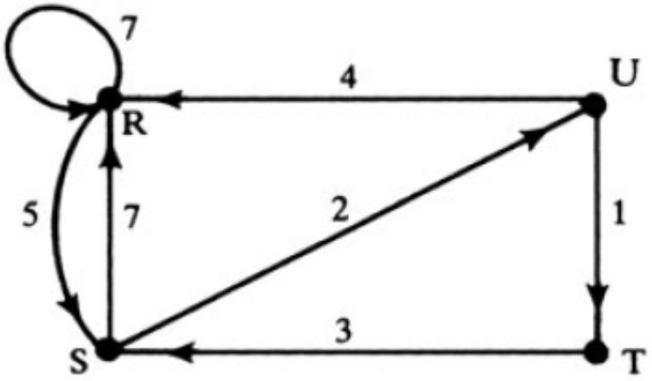
More exactly,

$$Q_k[i, j] = \text{MIN}(Q_{k-1}[i, j],\ Q_{k-1}[i, k] + Q_{k-1}[k, j])$$

The initial matrix Q_0 is the same as the weight matrix W except that each 0 in w is replaced by ∞ (or a very, very large number). The final matrix Q_m will be the desired matrix Q.

Shortest Path Algorithm

- Figure below shows a weighted graph G and its weight matrix W where we assume that v 1 = R, v 2 = S, v 3 = T, v 4 = U.
- Suppose we apply the modified Warshall's algorithm to our weighted graph G



$$W = \begin{bmatrix} 7 & 5 & 0 & 0 \\ 7 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 1 & 0 \end{bmatrix}$$

Shortest Path Algorithm

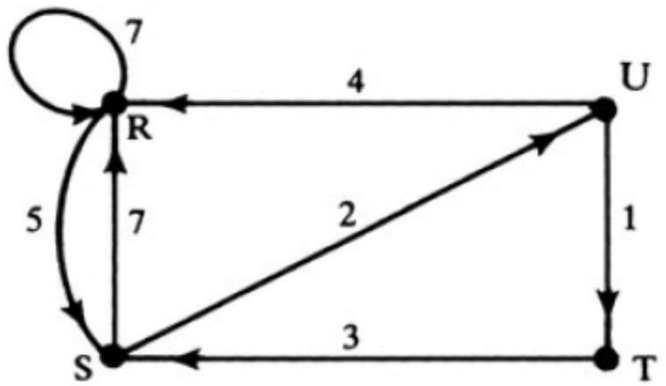
$$Q_0 = \begin{bmatrix} 7 & 5 & \infty & \infty \\ 7 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & \infty & 1 & \infty \end{bmatrix} \qquad \begin{bmatrix} RR & RS & — & — \\ SR & — & — & SU \\ — & TS & — & — \\ UR & — & UT & — \end{bmatrix}$$

$$Q_1 = \begin{bmatrix} 7 & 5 & \infty & \infty \\ 7 & 12 & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & (9) & 1 & \infty \end{bmatrix} \qquad \begin{bmatrix} RR & RS & — & — \\ SR & SRS & — & SU \\ — & TS & — & — \\ UR & URS & UT & — \end{bmatrix}$$

$$Q_2 = \begin{bmatrix} 7 & 5 & (\infty) & 7 \\ 7 & 12 & \infty & 2 \\ 10 & 3 & \infty & 5 \\ 4 & 9 & 1 & 11 \end{bmatrix} \qquad \begin{bmatrix} RR & RS & — & RSU \\ SR & SRS & — & SU \\ TSR & TS & — & TSU \\ UR & URS & UT & URS \end{bmatrix}$$

$$Q_3 = \begin{bmatrix} 7 & 5 & \infty & 7 \\ 7 & 12 & \infty & 2 \\ 10 & 3 & \infty & 5 \\ 4 & (4) & 1 & 6 \end{bmatrix} \qquad \begin{bmatrix} RR & RS & — & RSU \\ SR & SRS & — & SU \\ TSR & TS & — & TSU \\ UR & UTS & UT & UTSU \end{bmatrix}$$

$$Q_4 = \begin{bmatrix} 7 & 5 & 8 & 7 \\ 7 & 11 & 3 & 2 \\ (9) & 3 & 6 & 5 \\ 4 & 4 & 1 & 6 \end{bmatrix} \qquad \begin{bmatrix} RR & RS & RSUT & RSU \\ SR & SURS & SUT & SU \\ TSUR & TS & TSUT & TSU \\ UR & UTS & UT & UTSU \end{bmatrix}$$



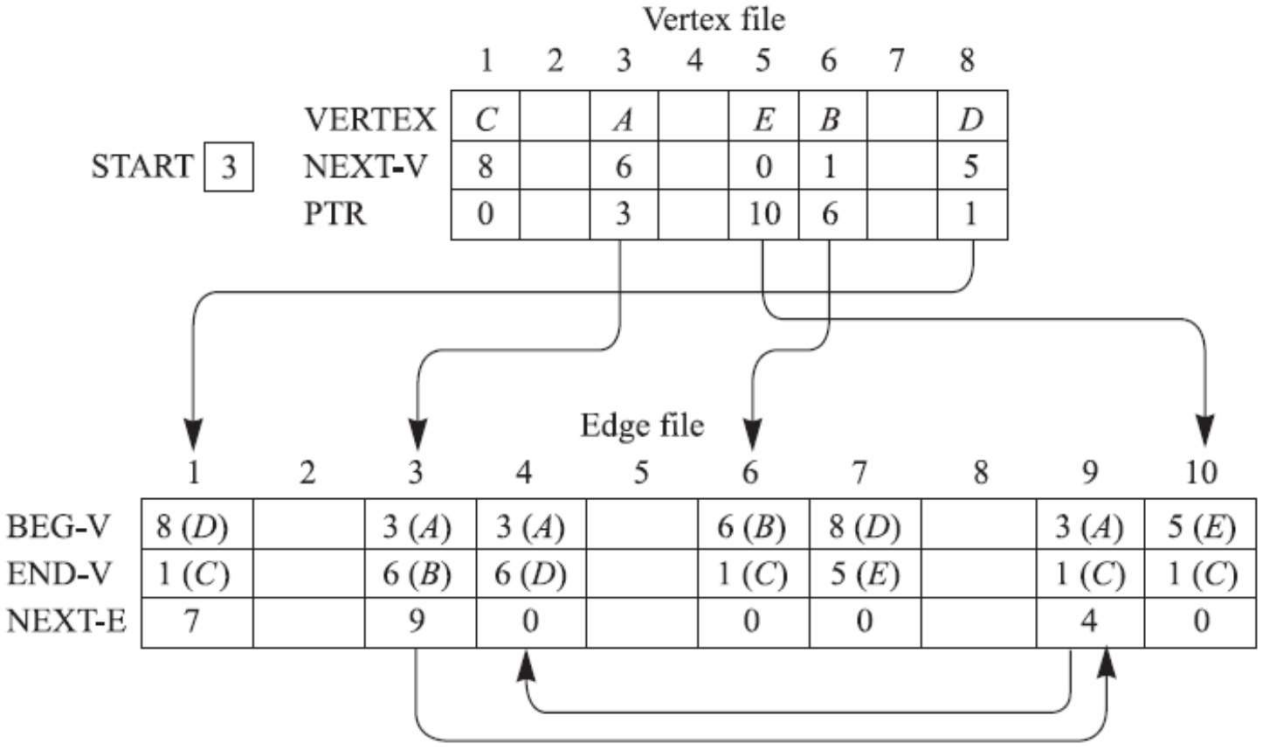
$$Q_1[4,2] = \text{MIN}(Q_0[4,2],\ Q_0[4,1] + Q_0[1,2]) = \text{MIN}(\infty, 4+5) = 9$$

$$Q_2[1,3] = \text{MIN}(Q_1[1,3],\ Q_1[1,2] + Q_1[2,3]) = \text{MIN}(\infty, 5+\infty) = \infty$$

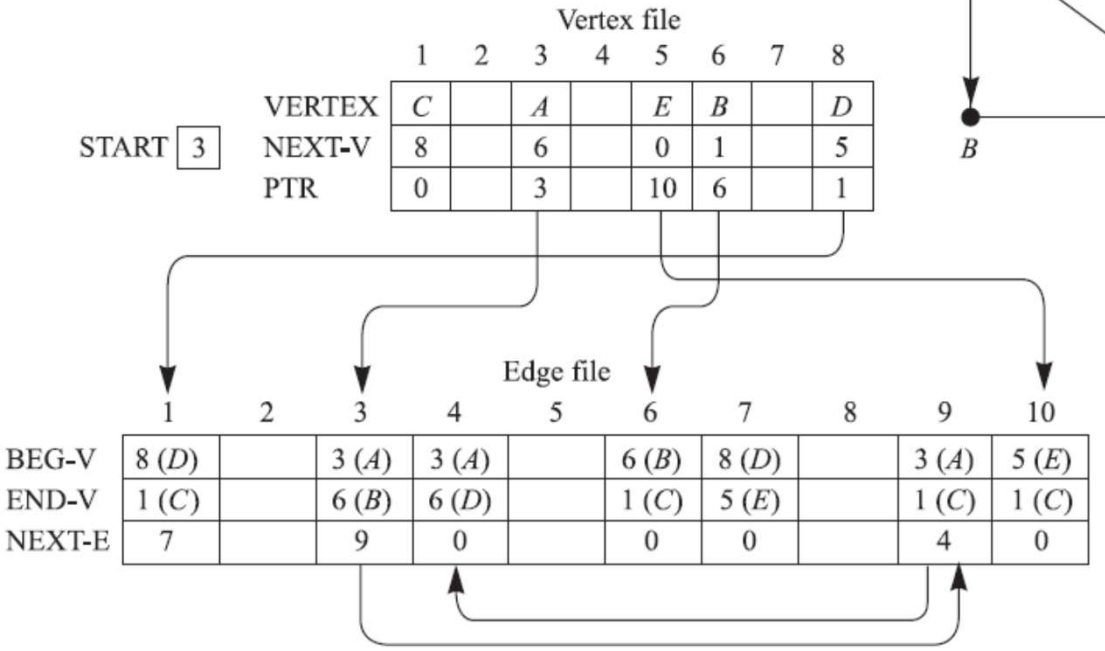
$$Q_3[4,2] = \text{MIN}(Q_2[4,2],\ Q_2[4,3] + Q_2[3,2]) = \text{MIN}(9, 3+1) = 4$$

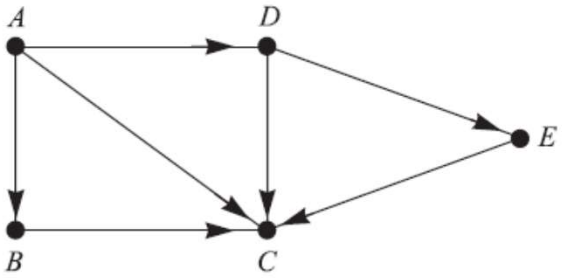
$$Q_4[3,1] = \text{MIN}(Q_3[3,1],\ Q_3[3,4] + Q_3[4,1]) = \text{MIN}(10, 5+4) = 9$$

Linked Representation of Directed Graphs



Linked Representation of Directed Graphs



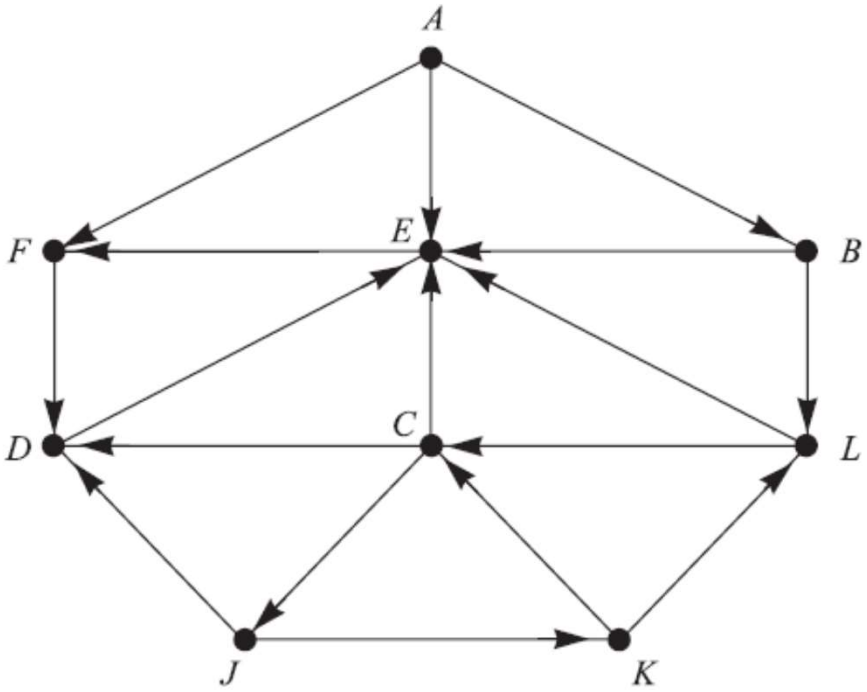


(*a*) Graph *G*

Vertex	Adjacency list
A	*B*, *C*, *D*
B	*C*
C	∅
D	*C*, *E*
E	*C*

(*b*) Adjacency lists of *G*

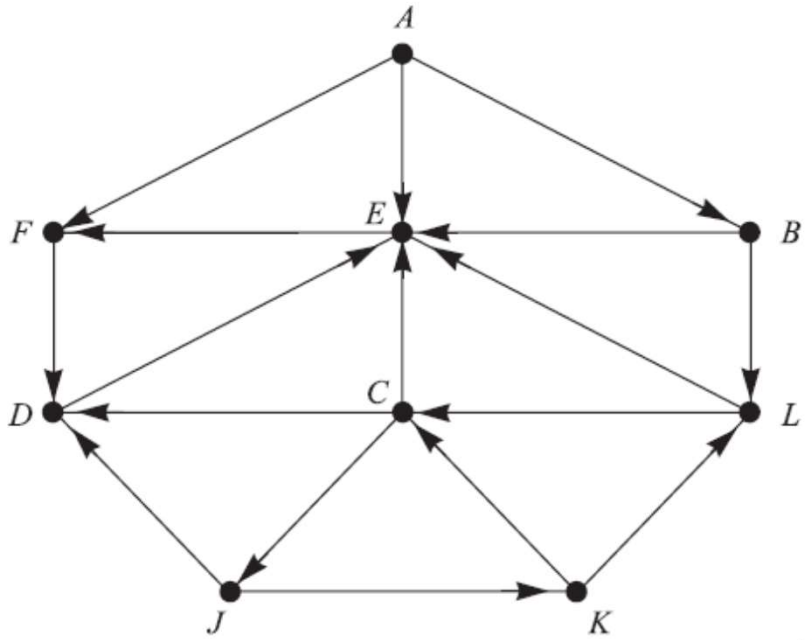
DFS for Directed Graphs



Adjacency lists
A: *B*, *E*, *F*
B: *E*, *L*
C: *D*, *E*, *J*
D: *E*
E: *F*
F: *D*
J: *D*, *K*
K: *C*, *L*
L: *C*, *E*

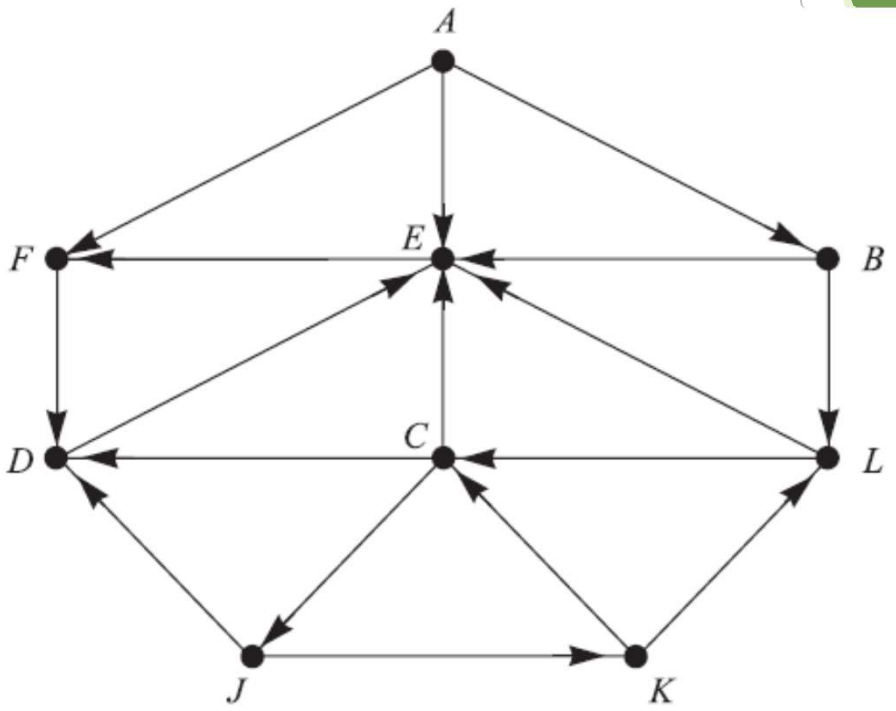
DFS for Directed Graphs

STACK	Vertex
J	J
K_J, D_J	K_J
L_K, C_K, D_J	L_K
$E_L, C_L, \not{C}_K, D_J$	E_L
F_E, C_L, D_J	F_E
$D_F, C_L, \not{D}_J$	D_F
C_L	C_L
$\varnothing$	

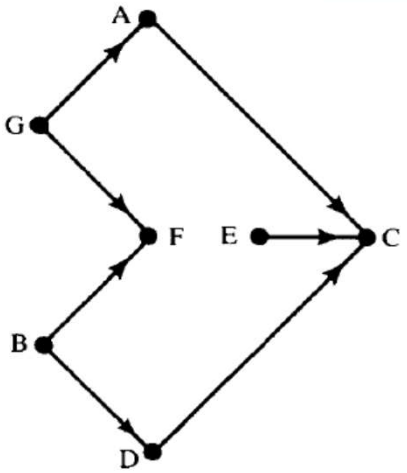


BFS for Directed Graphs

QUEUE	Vertex
A	A
F_A, E_A, B_A	B_A
L_B, F_A, E_A	E_A
L_B, F_A	F_A
D_F, L_B	L_B
C_L, D_F	D_F
C_L	C_L
J_C	J_C



DIRECTED CYCLE-FREE GRAPHS, TOPOLOGICAL SORT



Adjacency lists	
A:	C
B:	D, F
C:	
D:	C
E:	C
F:	
G:	A, F

Let S be a directed graph with the following two properties:

(1) Each vertex v_i of S represents a task.

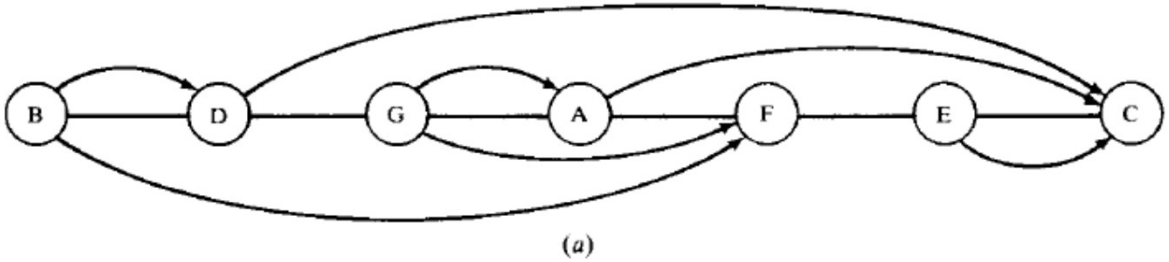
(2) Each (directed) edge (u, v) of S means that task u must be completed before beginning task v.

We note that such a graph S cannot contain a cycle, such as $P = (u, v, w, u)$, since, otherwise, we would have to complete u before beginning v, complete v before beginning w, and complete w before beginning u. That is, we cannot begin any of the three tasks in the cycle.

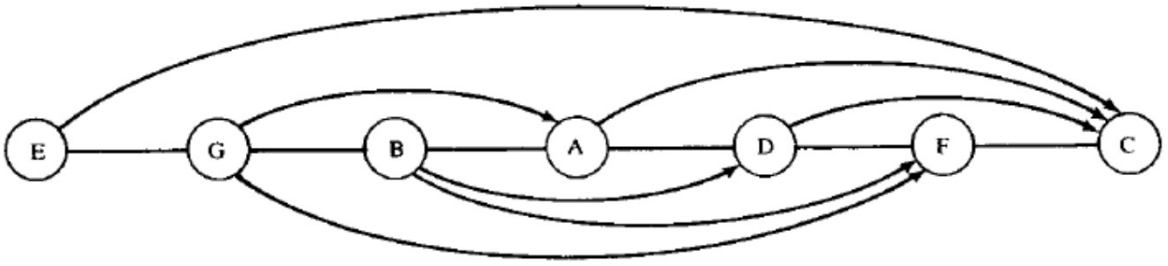
Such a graph S, which represents tasks and a prerequisite relation and which cannot have any cycles, is said to be *cycle-free* or *acyclic*. A directed acyclic (cycle-free) graph is called a *dag* for short. Figure 9-16 is an example of such a graph.

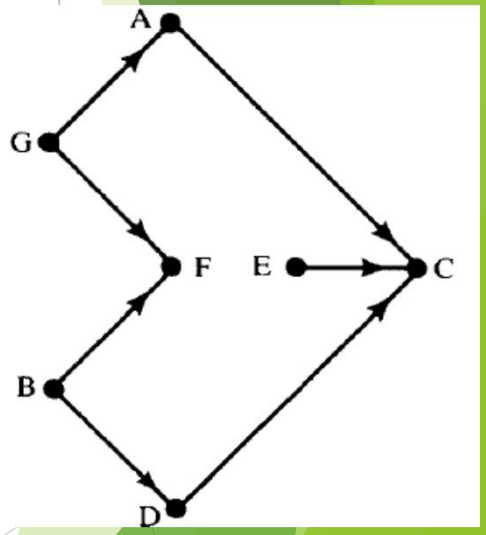
DIRECTED CYCLE-FREE GRAPHS, TOPOLOGICAL SORT

- A fundamental operation on a dag S is to process the vertices one after the other so that the vertex u is always processed before vertex v whenever (u,v) is an edge. Such a linear ordering T of the vertices of S, which may not be unique, is called a topological sort



(a)





DAGs

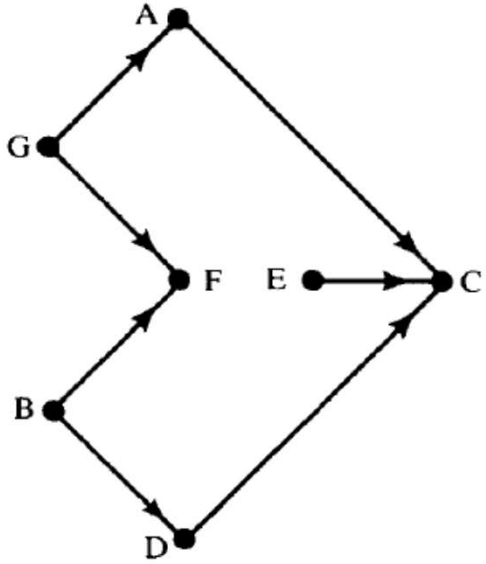
Theorem 9.8: Let S be a finite directed cycle-free graph. Then there exists a topological sort T of the graph S.

Note that the theorem states only that a topological sort exists. We now give an algorithm which will find a topological sort. The main idea of the algorithm is that any vertex (node) N with zero indegree may be chosen as the first element in the sort T. The algorithm essentially repeats the following two steps until S is empty:

(1) Find a vertex N with zero indegree.
(2) Delete N and its edges from the graph S.

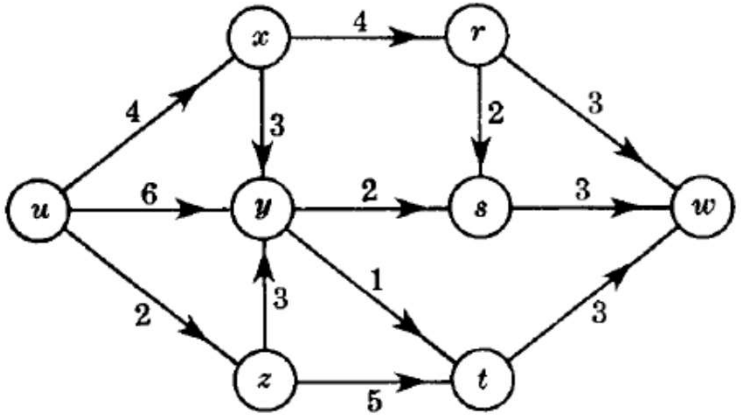
DAGs

QUEUE	*GEB*	*DGE*	*DG*	*FAD*	*FA*	*CF*	*C*	∅
Vertex	*B*	*E*	*G*	*D*	*A*	*F*	*C*	

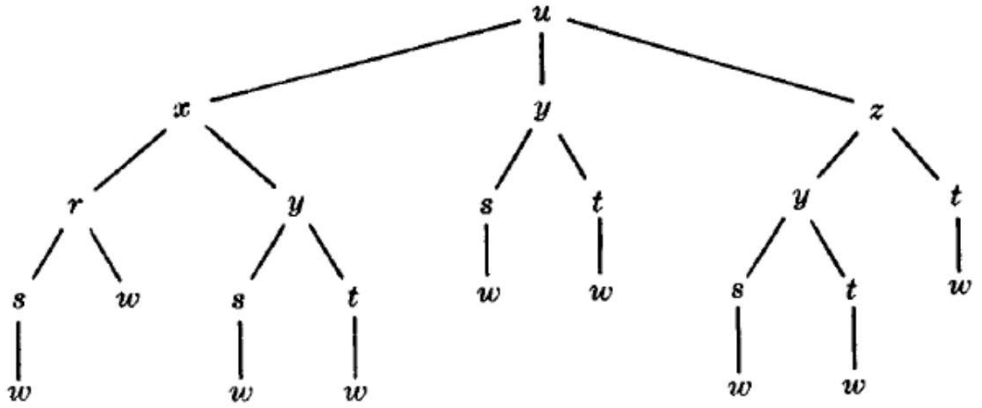


Pruning Algorithm

Let G be a weighted directed cycle-free graph. We seek the shortest path between two vertices, say u and w. We assume G is finite so at each step there is a finite number of moves. Since G is cycle-free, all paths between u and w can be given by a rooted tree with u as the root. Figure 9-19(b) enumerates all the paths between u and w in the graph in Fig. 9-19(a).



(a)



(b)

Pruning Algorithm

Pruning Algorithm

This algorithm finds the shortest path between a vertex u and a vertex w in a weighted directed cycle-free graph G. The algorithm has the following properties:

(a) During the algorithm each vertex v' of G is assigned two things:

(1) A number $\ell(v')$ denoting the current minimal length of a path from u to v'.

(2) A path $p(v')$ from u to v' of length $\ell(v')$.

(b) Initially, we set $\ell(u) = 0$ and $p(u) = u$. Every other vertex v is initially assigned $\ell(v) = \infty$ and $p(v) = \varnothing$.

(c) Each step of the algorithm examines an edge $e = (v', v)$ from v' to v with, say, length k. We calculate $\ell(v') + k$.

(1) Suppose $\ell(v') + k < \ell(v)$. Then we have found a shorter path from u to v. Thus we update:

$$\ell(v) = \ell(v') + k \quad \text{and} \quad p(v) = p(v')v$$

(This is always true when $\ell(v) = \infty$, that is, when we first enter the vertex v.)

(2) Otherwise, we do not change $\ell(v)$ and $p(v)$.

If no other unexamined edges enter v, we will say that $p(v)$ has been determined.

(d) The algorithm ends when $p(w)$ has been determined.

Remark: The edge $e = (v', v)$ in (c) can only be chosen if v' has been previously visited, that is, if $p(v')$ is not empty. Furthermore, it is usually best to examine an edge which begins at a vertex v' whose path $p(v')$ has been determined.

Pruning Algorithm

From u: The successive vertices are x, y, and z, which are all entered for the first time. Thus:

(1) set $\ell(x) = 4$, $p(x) = ux$.

(2) set $\ell(y) = 6$, $p(y) = uy$.

(3) set $\ell(z) = 2$, $p(z) = uz$.

Note that $p(x)$ and $p(z)$ have been determined.

From x: The successive vertices are r, entered for the first time, and y. Thus:

(1) Set $\ell(r) = 4 + 4 = 8$ and $p(r) = p(x)r = uxr$.

(2) We calculate:

$$\ell(x) + k = 4 + 3 = 7 \quad \text{which is not less than} \quad \ell(y) = 6.$$

Thus we leave $\ell(y)$ and $p(y)$ alone.

Note that $p(r)$ has been determined.

From z: The successive vertices are t, entered for the first time, and y. Thus:

(1) Set $\ell(t) = \ell(z) + k = 2 + 5 = 7$ and $p(t) = p(z)t = urt$.

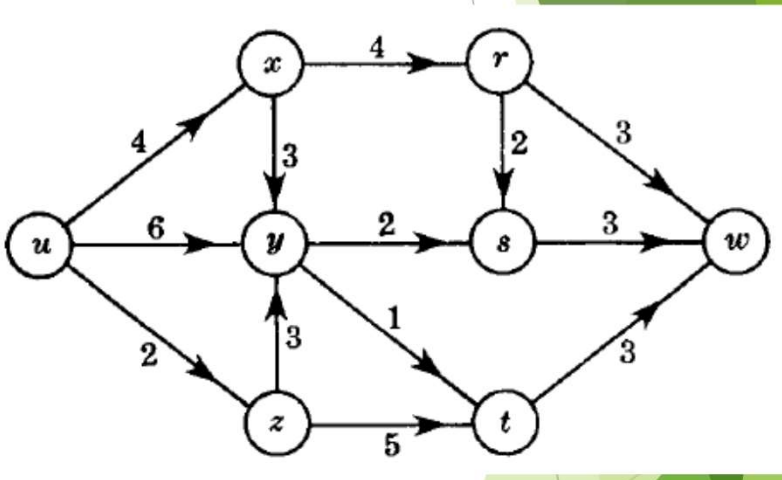
(2) We calculate:

$$\ell(z) + k = 2 + 3 = 5 \quad \text{which is less than} \quad \ell(y) = 6.$$

We have found a shorter path to y, and so we update $\ell(y)$ and $p(y)$; set:

$$\ell(y) = \ell(z) + k = 5 \quad \text{and} \quad p(y) = p(z)y = uzy$$

Now $p(y)$ has been determined.



From *y*: The successive vertices are s, entered for the first time, and t. Thus:

(1) Set $\ell(s) = \ell(y) + k = 5 + 2 = 7$ and $p(s) = p(y)s = uzys$.

(2) We calculate:

$$\ell(y) + k = 5 + 1 = 6 \quad \text{which is less than} \quad \ell(t) = 7.$$

Thus we change $\ell(t)$ and $p(t)$ to read:

$$\ell(t) = \ell(y) + 1 = 6 \quad \text{and} \quad p(t) = p(y)t = uzyt.$$

Now $p(t)$ has been determined.

From *r*: The successive vertices are w, entered for the first time, and s. Thus:

(1) Set $\ell(w) = \ell(r) + 3 = 11$ and $p(w) = p(r)w = uxrw$.

(2) We calculate:

$$\ell(r) + k = 8 + 2 = 10 \quad \text{which is less than} \quad \ell(s) = 7.$$

Thus we leave $\ell(s)$ and $p(s)$ alone.

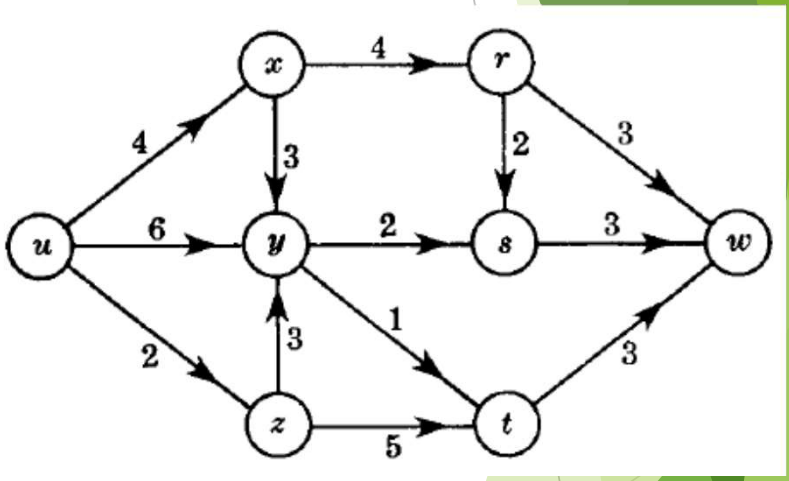
Note that $p(s)$ has been determined.

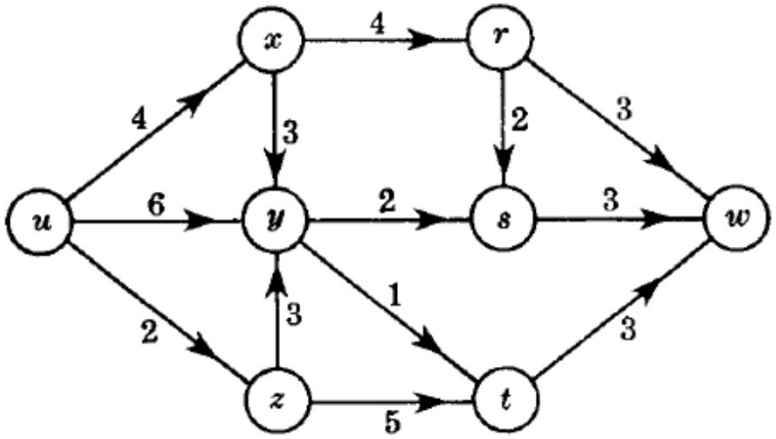
From *s*: The successive vertex is w. We calculate:

$$\ell(s) + k = 7 + 3 = 10 \quad \text{which is less than} \quad \ell(w) = 11.$$

Thus we change, $\ell(w)$ and $p(w)$ to read:

$$\ell(w) = \ell(s) + 3 = 10 \quad \text{and} \quad p(w) = p(s)w = uzysw.$$





From *t*: The successive vertex is w. We calculate:

$$\ell(t) + k = 6 + 3 = 9 \quad \text{which is less than} \quad \ell(w) = 10.$$

Thus we undate $\ell(w)$ and $p(w)$ as follows:

$$\ell(w) = \ell(t) + 3 = 9 \quad \text{and} \quad p(w) = p(t) = uzytw$$

Now $p(w)$ has been determined.

The algorithm is finished since $p(w)$ has been determined. Thus $p(w) = uzytw$ is the shortest path from u to w and $\ell(w) = 9$.

Binary Trees

Binary Trees

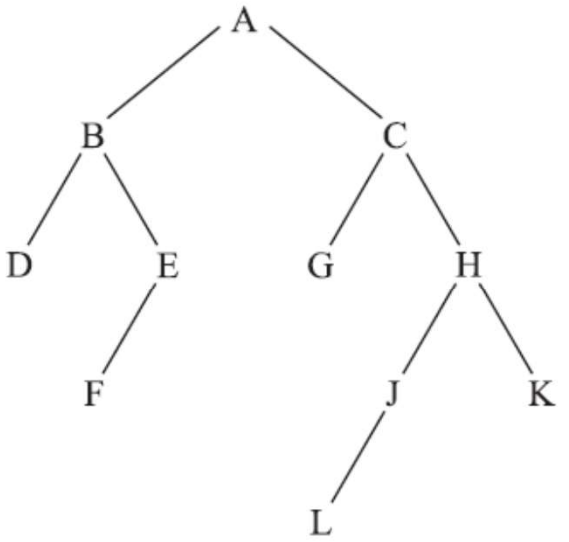
A *binary tree* T is defined as a finite set of elements, called *nodes*, such that:

(1) T is empty (called the *null tree* or *empty tree*), or
(2) T contains a distinguished node R, called the *root* of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2.

If T does contain a root R, then the two trees T_1 and T_2 are called, respectively, the left and right subtrees of R. If T_1 is nonempty, then its root is called the *left successor* of R; similarly, if T_2 is nonempty, then its root is called the *right successor* of R.

The above definition of a binary tree T is recursive since T is defined in terms of the binary subtrees T_1 and T_2. This means, in particular, that every node N of T contains a left and a right subtree, and either subtree or both subtrees may be empty. Thus every node N in T has 0, 1, or 2 successors. A node with no successors is called a *terminal* node. Thus both subtrees of a terminal node are empty.

Binary Trees



A binary tree T is frequently presented by a diagram in the plane called a *picture* of T. Specifically, the diagram in Fig. 10-1(a) represents a binary tree as follows:

(i) T consists of 11 nodes, represented by the letters A through L, excluding I.

(ii) The root of T is the node A at the top of the diagram.

(iii) A left-downward slanted line at a node N indicates a left successor of N; and a right-downward slanted line at N indicates a right successor of N.

(a) B is a left successor and C is a right successor of the root A.

(b) The left subtree of the root A consists of the nodes B, D, E, and F, and the right subtree of A consists of the nodes C, G, H, J, K, and L.

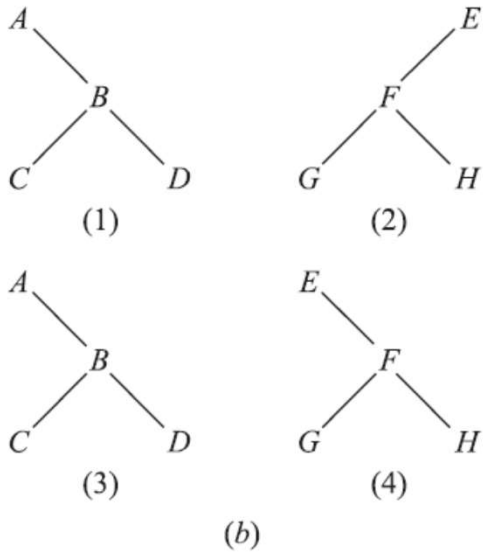
(c) The nodes A, B, C, and H have two successors, the nodes E and J have only one successor, and the nodes D, F, G, L, and K have no successors, i.e., they are terminal nodes.

Similar Trees

Similar Binary Trees

Binary trees T and T' are said to be *similar* if they have the same structure or, in other words, if they have the same shape. The trees are said to be *copies* if they are similar and if they have the same contents at corresponding nodes.

EXAMPLE 10.1 Consider the four binary trees in Fig. 10-1(*b*). The three trees (1), (3), and (4) are similar. In particular the trees (1) and (3) are copies since they also have the same data at corresponding nodes. The tree (2) is neither similar nor a copy of the tree (4) because, in a binary tree, we distinguish between a left successor and a right successor even when there is only one successor.



Terminology

Terminology

Terminology describing family relationships is frequently used to describe relationships between the nodes of a tree T. Specifically, suppose N is a node in T with left successor S_1 and right successor S_2. Then N is called the *parent* (or *father*) of S_1 and S_2. Analogously, S_1 is called the *left child* (or *left son*) of N, and S_2 is called the *right child* (or *right son*) of N. Furthermore, S_1 and S_2 are said to be *siblings* (or *brothers*). Every node N in a binary tree T, except the root, has a unique parent, called the *predecessor* of N.

The terms descendant and ancestor have their usual meaning. That is, a node L is called a *descendant* of a node N (and N is called an *ancestor* of L) if there is a succession of children from N to L. In particular, L is called a *left* or *right descendant* of N according to whether L belongs to the left or right subtree of N.

Terminology from graph theory and horticulture are also used with a binary tree T. Specifically, the line drawn from a node N of T to a successor is called an *edge*, and a sequence of consecutive edges is called a *path*. A terminal node is called a *leaf*, and a path ending in a leaf is called a *branch*.

Each node in a binary tree T is assigned a *level number*, as follows. The root R of the tree T is assigned the level number 0, and every other node is assigned a level number which is 1 more than the level number of its parent. Furthermore, those nodes with the same level number are said to belong to the same *generation*.

The *depth* (or *height*) of a tree T is the maximum number of nodes in a branch of T. This turns out to be 1 more than the largest level number of T. The tree T in Fig. 10-1(a) has depth 5.

Complete binary trees

Consider any binary tree T. Each node of T can have at most two children. Accordingly, one can show that level r of T can have at most 2^r nodes. The tree T is said to be *complete* if all its levels, except possibly the last, have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible. Thus there is a unique complete tree T_n with exactly n nodes (where we ignore the contents of the nodes). The complete tree T_{26} with 26 nodes appears in Fig. 10-2.

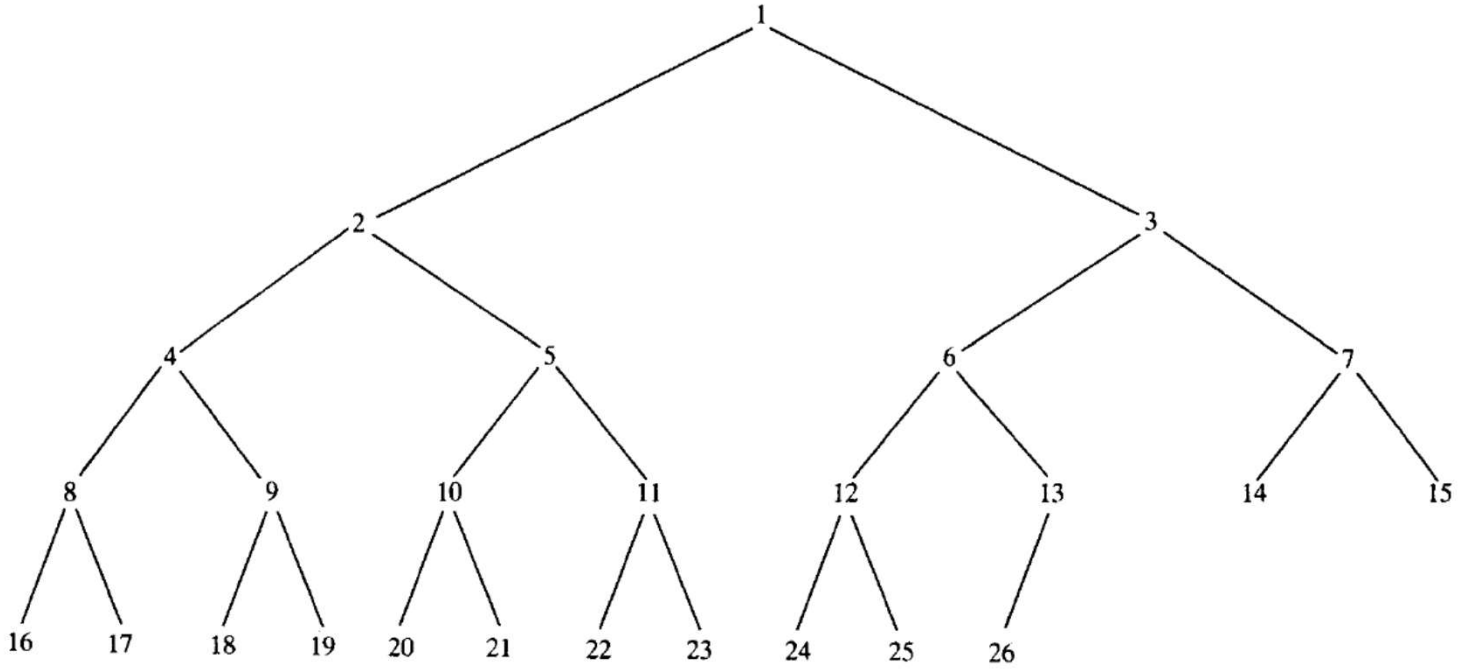
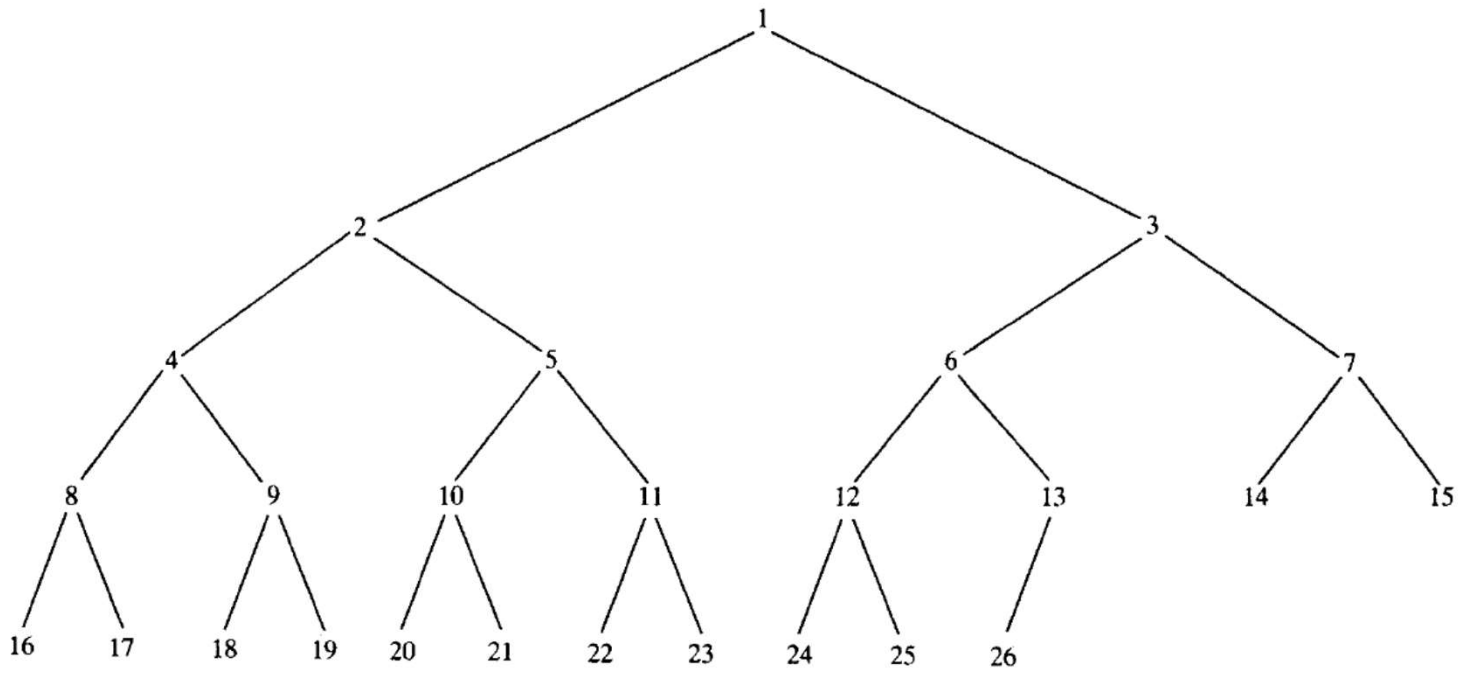


Fig. 10-2 Complete tree T_{26}



The nodes of the complete binary tree T_{26} in Fig. 10-2 have been purposely labeled by the integers 1, 2, . . . , 26, from left to right, generation by generation. With this labeling, one can easily determine the children and parent of any node K in any complete tree T_n. Specifically, the left and right children of the node K are, respectively, 2^*K and $2^*K + 1$, and the parent of K is the node $[K/2]$. For example, the children of node 9 are the nodes 18 and 19, and its parent is the node $[9/2] = 4$. The depth d_n of the complete tree T_n with n nodes is given by

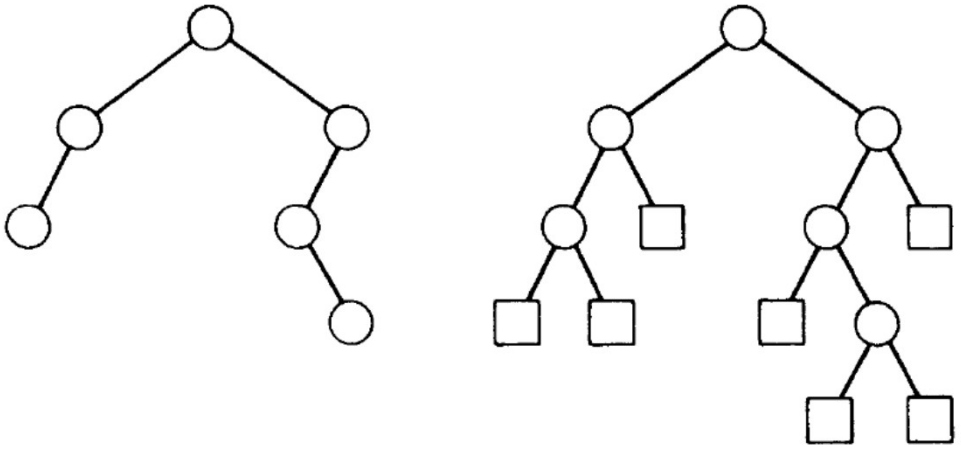
$$d_n = \lfloor \log_2 n + 1 \rfloor$$

This is a relatively small number. For example, if the complete tree T_n has $n = 1\,000\,000$ nodes, then its depth $d_n = 21$.

Extended Binary Trees: 2-Trees

A binary tree tree T is said to be a *2-tree* or an *extended binary tree* if each node N has either 0 or 2 children. In such a case, the nodes with two children are called *internal nodes*, and the nodes with 0 children are called *external nodes*. Sometimes the nodes are distinguished in diagrams by using circles for internal nodes and squares for external nodes.

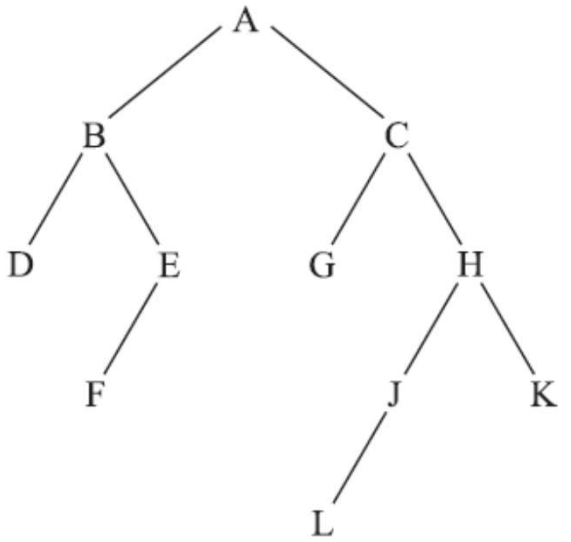
The term "extended binary tree" comes from the following operation. Consider any binary tree T, such as the tree in Fig. 10-3(a). Then T may be "converted" into a 2-tree by replacing each empty subtree by a new node, as pictured in Fig. 10-3(b). Observe that the new tree is, indeed, a 2-tree. Furthermore, the nodes in the original tree T are now the internal nodes in the extended tree, and the new nodes are the external nodes in the extended tree. We note that if a 2-tree has n internal nodes, then it will have $n + 1$ external nodes.



(a) Binary tree T.

(b) Extended 2-tree.

Linked Representation of Binary Trees



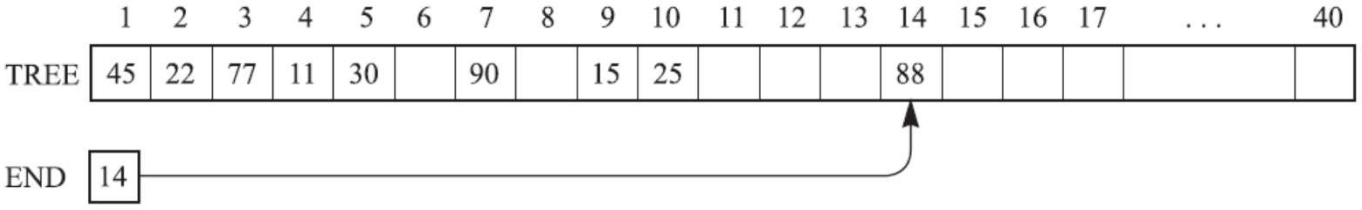
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
INFO	*K*	*C*	*G*		*A*	*H*	*L*			*B*		*F*	*E*			*J*	*D*	
LEFT	0	3	0		10	16	0			17		0	12			7	0	
RIGHT	0	6	0		2	1	0			13		0	0			0	0	

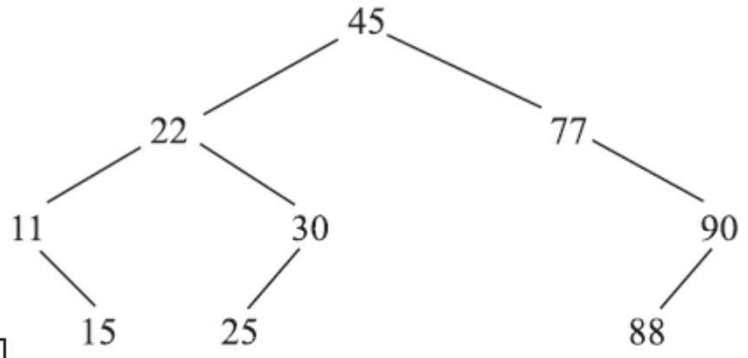
ROOT 5

Sequential Representation

Suppose T is a binary tree that is complete or nearly complete. Then there is an efficient way of maintaining T in memory called the *sequential representation* of T. This representation uses only a single linear array TREE together with a pointer variable END as follows:

(1) The root R of T is stored in TREE[1].
(2) If a node N occupies TREE[K], then its left child is stored in TREE[2*K] and its right child is stored in TREE[2*K + 1].
(3) END contains the location of the last node of T.





Traversing Binary Trees

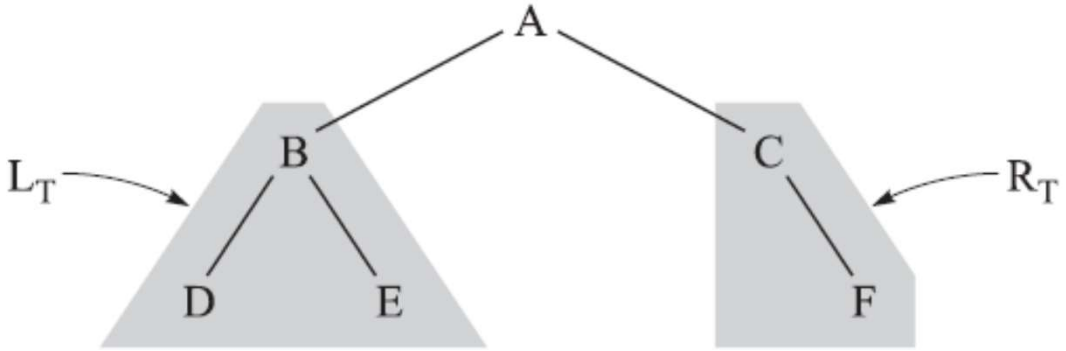
There are three standard ways of traversing a binary tree T with root R. These three algorithms, called *preorder*, *inorder*, and *postorder*, are as follows:

Preorder: (1) Process the root R.
(2) Traverse the left subtree of R in preorder.
(3) Traverse the right subtree of R in preorder.

Inorder: (1) Traverse the left subtree of R in inorder.
(2) Process the root R.
(3) Traverse the right subtree of R in inorder.

Postorder: (1) Traverse the left subtree of R in postorder.
(2) Traverse the right subtree of R in postorder.
(3) Process the root R.

Traversing Binary Trees



EXAMPLE 10.3 Consider the binary tree T in Fig. 10-7(a). Observe that A is the root of T, that the left subtree L_T of T consists of nodes B, D, and E, and the right subtree R_T of T consists of nodes C and F.

(a) The preorder traversal of T processes A, traverses L_T, and traverses R_T. However, the preorder traversal of L_T processes the root B, and then D and E; and the preorder traversal of R_T processes the root C and then F. Thus $ABDECF$ is the preorder traversal of T.

(b) The inorder traversal of T traverses L_T processes A, and traverses R_T. However, the inorder traversal of L_T processes D, B, and then E; and the inorder traversal of R_T processes C and then F. Thus $DBEACF$ is the inorder traversal of T.

(c) The postorder traversal of T traverses L_T, traverses R_T, and processes A. However, the postorder traversal of L_T, processes D, E, and then B, and the postorder traversal of RT processes F and then C. Accordingly, $DEBFCA$ is the postorder traversal of T.

Binary Search Trees

Although each node in a binary search tree may contain an entire record of data, the definition of the tree depends on a given field whose values are distinct and may be ordered.

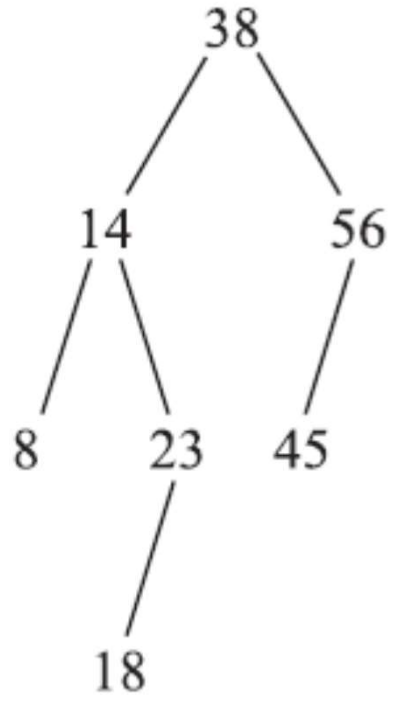
Definition: Suppose T is a binary tree. Then T is called a *binary search tree* if each node N of T has the following property:

> The value of N is greater than every value in the left subtree of N and is less than every value in the right subtree of N.

It is not difficult to see that the above property guarantees that the inorder traversal of T will yield a sorted listing of the elements of T.

Remark: The above definition of a binary search tree assumes that all the node values are distinct. There is an analogous definition of a binary search tree T which admits duplicates, that is, in which each node N has the following properties:

(a) $N > M$ for every node M in a left subtree of N.
(b) $N \leq M$ for every node M in a right subtree of N.



Binary Search Tree

Searching and Inserting in a Binary Search Tree

A search and insertion algorithm in a binary search tree T appears in Fig. 10-9.

Algorithm 10.1: A binary search tree T and an ITEM of information is given. The algorithm finds the location of ITEM in T, or inserts ITEM as a new node in the tree.

Step 1. Compare ITEM with the root N of the tree.
- (a) If ITEM $< N$, proceed to the left child of N.
- (b) If ITEM $> N$, proceed to the right child of N.

Step 2. Repeat Step 1 until one of the following occurs:
- (a) We meet a node N such that ITEM $= N$. In this case the search is successful.
- (b) We meet an empty subtree, which indicates the search is unsuccessful. Insert ITEM in place of the empty subtree.

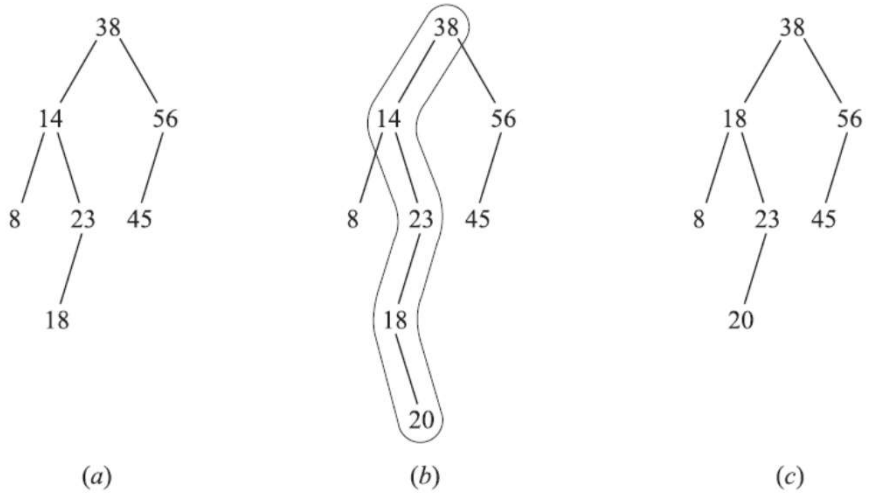
Step 3. Exit.

Binary Search Tree

EXAMPLE 10.6 Consider the binary search tree T in Fig. 10-8(a). Suppose ITEM = 20 is given, and we want to find or insert ITEM into T. Simulating Algorithm 10-1, we obtain the following steps:

(1) Compare ITEM = 20 with root $R = 38$. Since $20 < 38$, proceed to the left child of 38, which is 14.
(2) Compare ITEM = 20 with 14. Since $20 > 14$, proceed to the right child of 14, which is 23.
(3) Compare ITEM = 20 with 23. Since $20 < 23$, proceed to the left child of 23, which is 18.
(4) Compare ITEM = 20 with 18. Since $20 > 18$ and 18 has no right child, insert 20 as the right child of 18.

Figure 10-11(b) shows the new tree with ITEM = 20 inserted. The path down the tree during the algorithm has been ringed.



Heaps

Heaps

Suppose H is a complete binary tree with n elements. We assume H is maintained in memory using its sequential representation, not a linked representation. (See Section 10.4.)

Definition 10.1: Suppose H is a complete binary tree. Then H is called a *heap*, or a *maxheap*, if each node N has the following property.

> The value of N is greater than or equal to the value at each of the children of N.

Accordingly, in a heap, the value of N exceeds the value of every one of its descendants. In particular, the root of H is a largest value of H.

A *minheap* is defined analogously: The value of N is less than or equal to the value at each of its children.

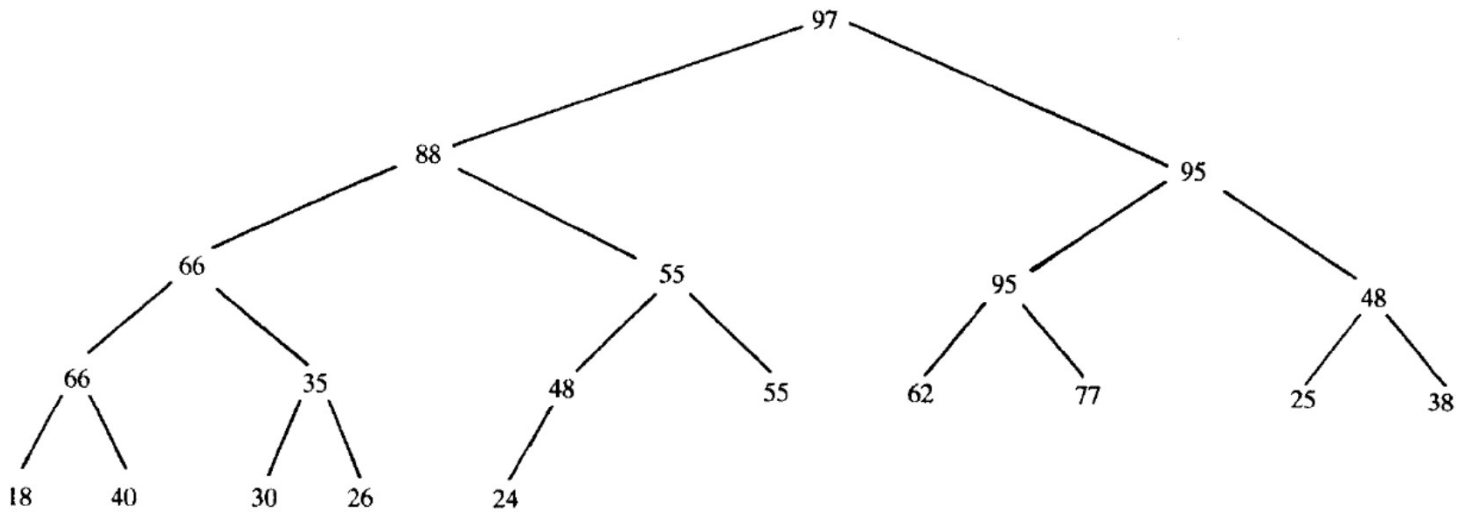
Inserting into a Heap

Algorithm 10.3: *A* heap H and a new ITEM are given. The algorithm inserts ITEM into H.

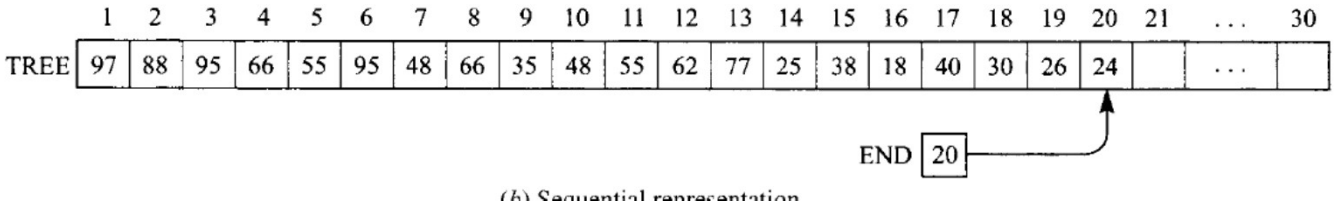
Step 1. Adjoin ITEM at the end of H so that H is still a complete tree, but not necessarily a heap.

Step 2. (Reheap) Let ITEM rise to its "appropriate place" in H so that H is a heap. That is:
- (a) Compare ITEM with its parent P(ITEM). If ITEM $>$ P(ITEM), then interchange ITEM and P(ITEM).
- (b) Repeat (a) until ITEM $\leq$ P(ITEM).

Step 3. Exit.



(*a*) Heap



(*b*) Sequential representation

EXAMPLE 10.9 Consider the heap H in Fig. 10-11. Suppose we want to insert ITEM = 70 into H. Simulating Algorithm 10.3, we first adjoin ITEM as the last element of the complete tree; that is, as the right child of 48.

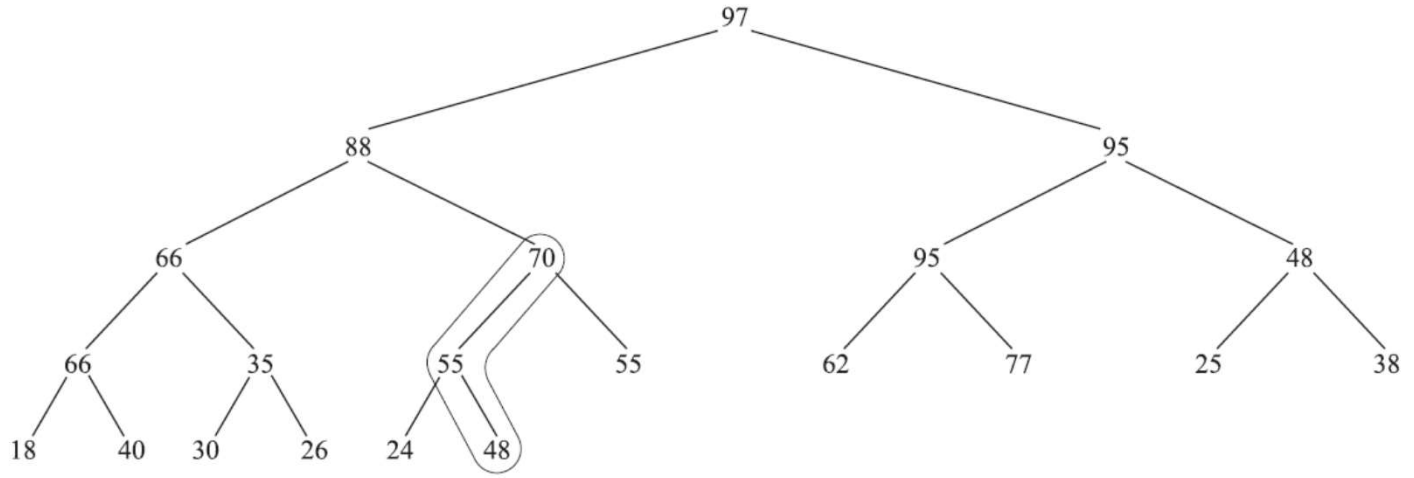
In other words, we set TREE[21] = 70 and END = 21. Then we reheap, i.e., we let ITEM rise to its appropriate place as follows:

(a) Compare ITEM = 70 with its parent 48. Since 70 > 48, we interchange 70 and 48.

(b) Compare ITEM = 70 with its new parent 55. Since 70 > 55, we interchange 70 and 55.

(c) Compare ITEM = 70 with its parent 88. Since 70 < 88, ITEM = 70 has risen to its appropriate place in the heap H.

Figure 10-13 shows the final tree H with ITEM = 70 inserted. The path up the tree by ITEM has been circled.



Deleting the Root

Algorithm 10.4: The algorithm deletes the root R from a given heap H.

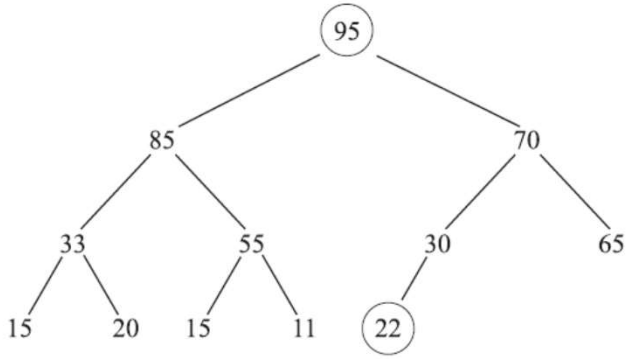
Step 1. Assign the root R to some variable ITEM.

Step 2. Replace the deleted root R by the last node of L of H so that H is still a complete binary tree, but not necessarily a heap. [That is, set TREE[1] := TREE[END] and then set END := END−1.]

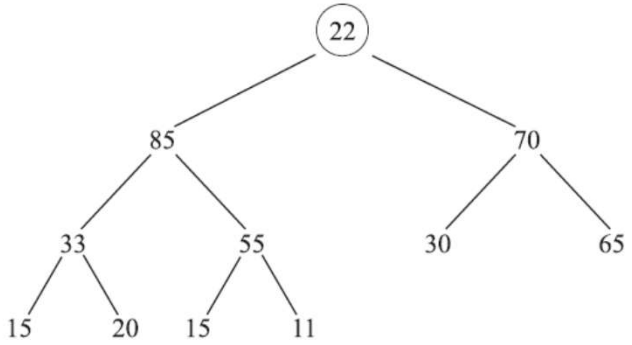
Step 3. (Reheap) Let L sink to its "appropriate place" in H so that H is a heap. That is:
- (a) Find the larger child LARGE(L) of L. If $L <$ LARGE(L), then interchange L and LARGE(L).
- (b) Repeat (a) until $L \geq$ LARGE(L).

Step 4. Exit.

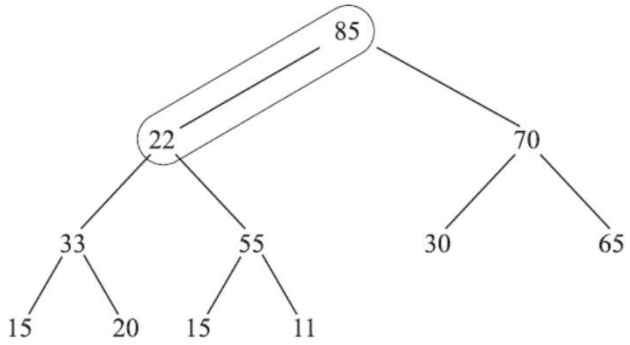
EXAMPLE 10.10 Consider the heap H in Fig. 10-15(a), where $R = 95$ is the root and $L = 22$ is the last node of H. Suppose we want to delete $R = 95$ from the heap H. Simulating Algorithm 10.4, we first "delete" $R = 95$ by assigning ITEM $= 95$, and then we replace $R = 95$ by $L = 22$. This yields the complete tree in Fig. 10-15(b)

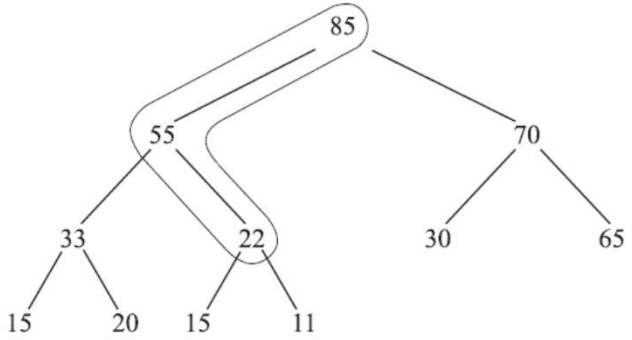


(a)



(b)





Path Length

Weighted Path Lengths

Suppose T is a 2-tree with n external nodes, and suppose each external node is assigned a (nonnegative) weight. The weighted path length (or simply path length) P of the tree T is defined to be the sum

$$P = W_1L_1 + W_2L_2 + \cdots + W_nL_n$$

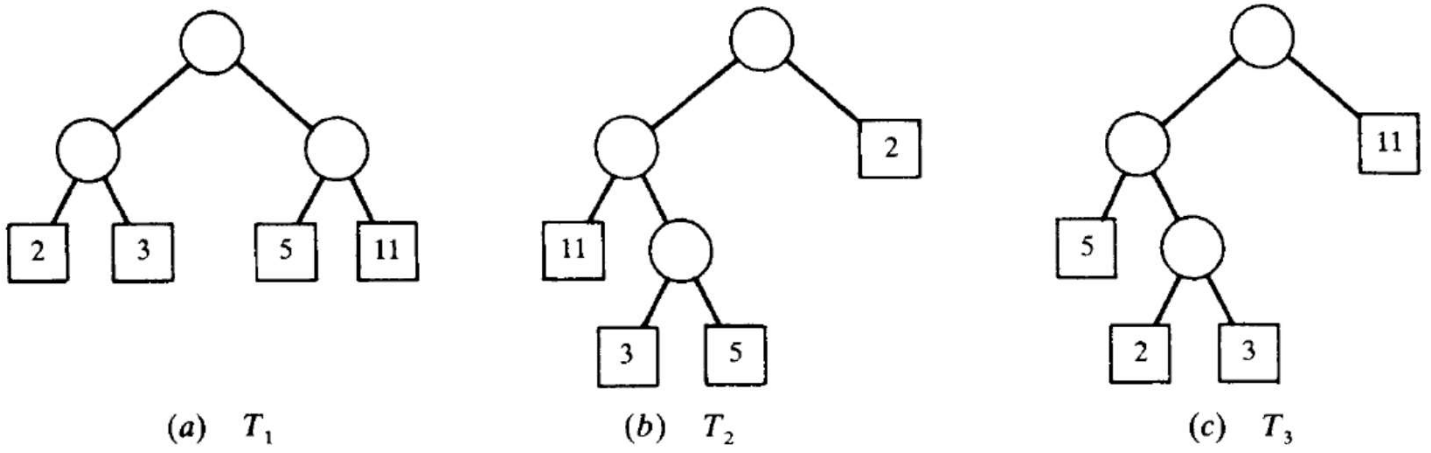
where W_1 is the weight at an external node N_i, and L_i is the length of the path from the root R to the node L_i. (The path length P exists even for nonweighted 2-trees where one simply assumes the weight 1 at each external node.)

Path Length

EXAMPLE 10.11 Figure 10-16 shows three 2-trees, T_1, T_2, T_3, each having external nodes with the same weights 2, 3, 5, and 11. The weighted path lengths of the three trees are as follows:

$$P_1 = 2(2) + 3(2) + 5(2) + 11(2) = 42$$
$$P_2 = 2(1) + 3(3) + 5(3) + 11(2) = 48$$
$$P_3 = 2(3) + 3(3) + 5(2) + 11(1) = 36$$

The quantities P_1 and P_3 indicate that the complete tree need not give a minimum path, and that the quantities P_2 and P_3 indicate that similar trees need not give the same path length.



(*a*) T_1 (*b*) T_2 (*c*) T_3

Huffman's Algorithm

The general problem we want to solve is the following. Suppose a list of n weights is given:

$$W_1, W_2, \ldots, W_n$$

Among all the 2-trees with n external nodes and with the given n weights, find a tree T with a minimum weighted path length. (Such a tree T is seldom unique.) Huffman gave an algorithm to find such a tree T.

Huffman's algorithm, which appears in Fig. 10-17, is recursively defined in terms of the number n of weights. In practice, we use an equivalent iterated form of the Huffman algorithm which constructs the desired tree T from the bottom up rather than from the top down.

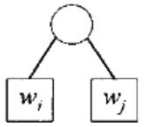
Huffman's Algorithm

Algorithm 10.5 (Huffman): The algorithm recursively finds a weighted 2-tree T with n given weights $w_1, w_2, \ldots, w_n$ which has a minimum weighted path length.

Step 1. Suppose $n = 1$. Let T be the tree with one node N with weight w_1, and then Exit.

Step 2. Suppose $n > 1$.

- (a) Find two minimum weights, say w_i and w_j, among the n given weights.
- (b) Replace w_i and w_j in the list by $w_i + w_j$, so the list has $n - 1$ weights.
- (c) Find a tree T' which gives a minimum weighted path length for the $n - 1$ weights
- (d) In the tree T', replace the external node

$\boxed{w_i + w_j}$ by the subtree 

- (e) Exit.

Huffman's Algorithm

EXAMPLE 10.12 Let A, B, C, D, E, F, G, H be eight data items with the following assigned weights:

Data item:	A	B	C	D	E	F	G	H
Weight:	22	5	11	19	2	11	25	5

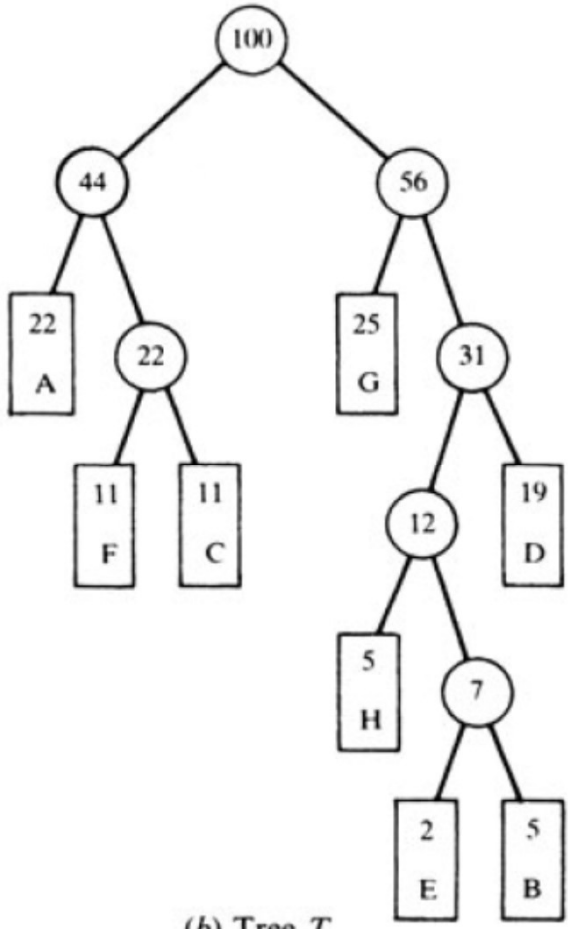
Construct a 2-tree T with a minimum weighted path length P using the above data as external nodes.

Apply the Huffman algorithm. That is, repeatedly combine the two subtrees with minimum weights into a single subtree as shown in Fig. 10-18(a). For clarity, the original weights are underlined, and a circled number indicates the root of a new subtree. The tree T is drawn from Step (8) backward yielding Fig. 10-18(b). (When splitting a node into two parts, we have drawn the smaller node on the left.) The path length P follows:

$$P = 22(2) + 11(3) + 11(3) + 25(2) + 5(4) + 2(5) + 5(5) + 19(3) = 280$$

(1) 22, 5, 11, 19, 2, 11, 25, 5
(2) 22, 11, 19, (7), 11, 25, 5
(3) 22, 11, 19, 11, 25, (12)
(4) 22, 19, (22), 25, 12
(5) 22, (31), 22, 25
(6) 31, (44), 25
(7) 44, (56)
(8) (100)

(*a*) Huffman algorithm



(*b*) Tree T

Questions?