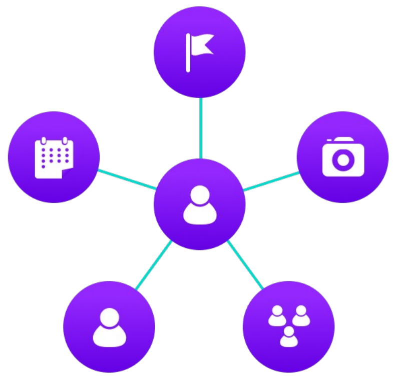
Graph

- A graph data structure is a collection of nodes that have data and are connected to other nodes.
- More precisely, a graph is a data structure (V, E) that consists of
- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u,v)

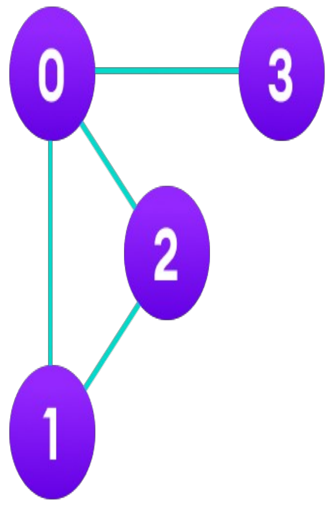
Example

- Let's try to understand this through an example. On facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node.
- Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.



Graph

- V = {0, 1, 2, 3}
- E = {(0,1), (0,2), (0,3), (1,2)}
- G = {V, E}



Terminology

- **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
- **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
- **Directed Graph**: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

Applications

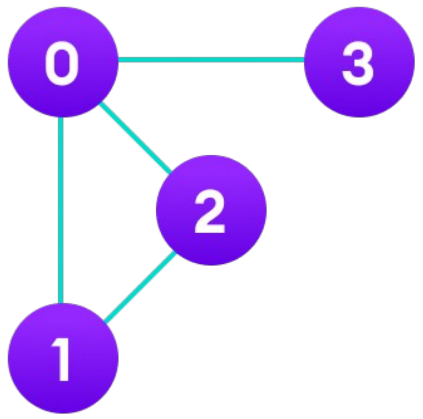
- Graphs are used to solve many real-life problems.
- Graphs are used to represent networks.
- The networks may include paths in a city or telephone network or circuit network.
- Graphs are also used in social networks like linkedIn, Facebook.
- For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

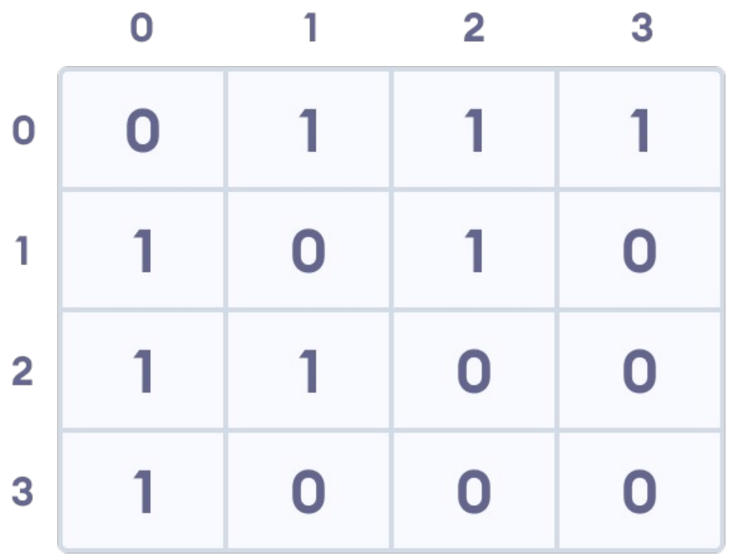
Graph Representation

- Graphs are commonly represented in two ways:
- 1. Adjacency Matrix
- 2. Adjacency List

Adjacency Matrix

- An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.
- If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

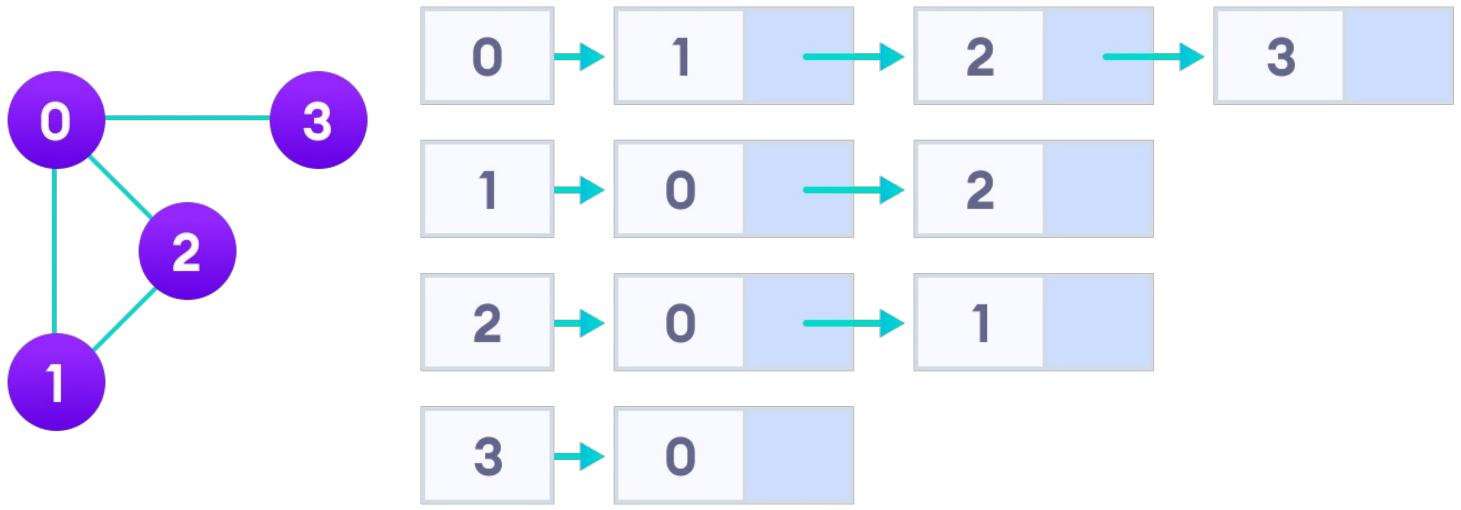




	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Adjacency List

- An adjacency list represents a graph as an array of linked lists.
- The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.
- The adjacency list for the graph we made in the first example is as follows:



Graph Operations

- The most common graph operations are:
- Check if the element is present in the graph
- Graph Traversal
- Add elements(vertex, edges) to graph
- Finding the path from one vertex to another

- A directed graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are directed from one vertex to another. A directed graph is sometimes called a digraph or a directed network. In contrast, a graph where the edges are bidirectional is called an undirected graph.a

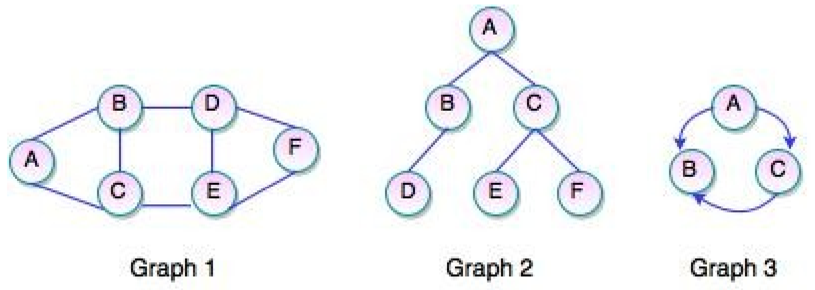


Fig. Graphs

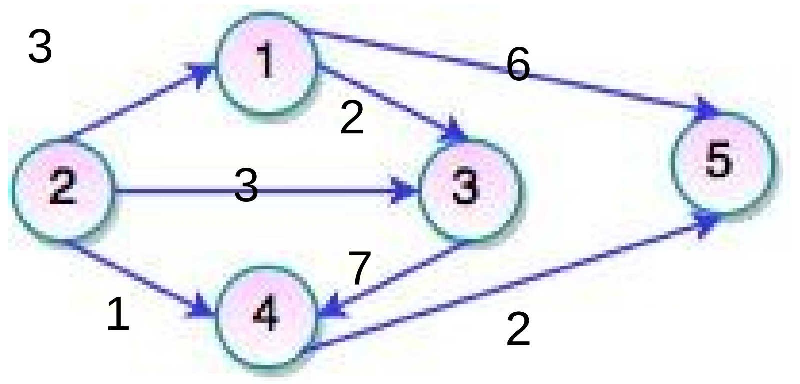
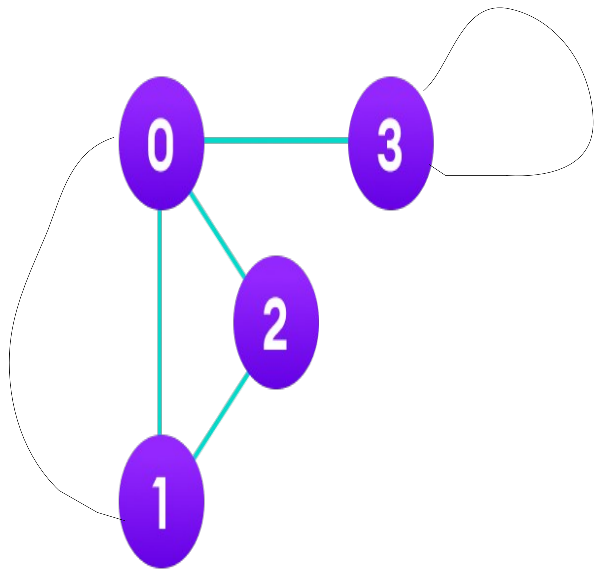


Fig. Directed Graph

{(2,1), (1,5), (1,3), (2,4), (2,3), (3,4), (4,5)}


0
3
2
1

- A directed graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are directed from one vertex to another. A directed graph is sometimes called a digraph or a directed network. In contrast, a graph where the edges are bidirectional is called an undirected graph.a

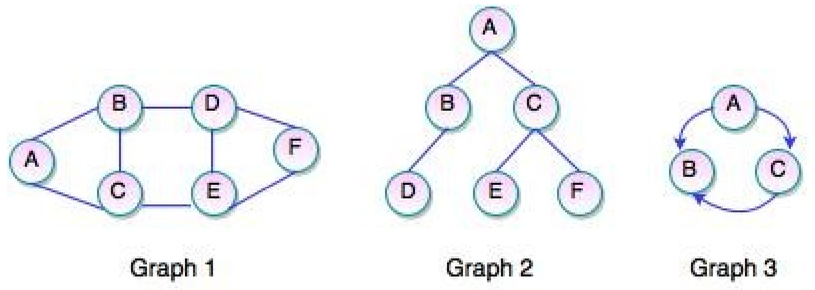


Fig. Graphs

Graph Traversal Algorithm

Traversing the graph means examining all the nodes and vertices of the graph.

There are two standard methods by using which, we can traverse the graphs.

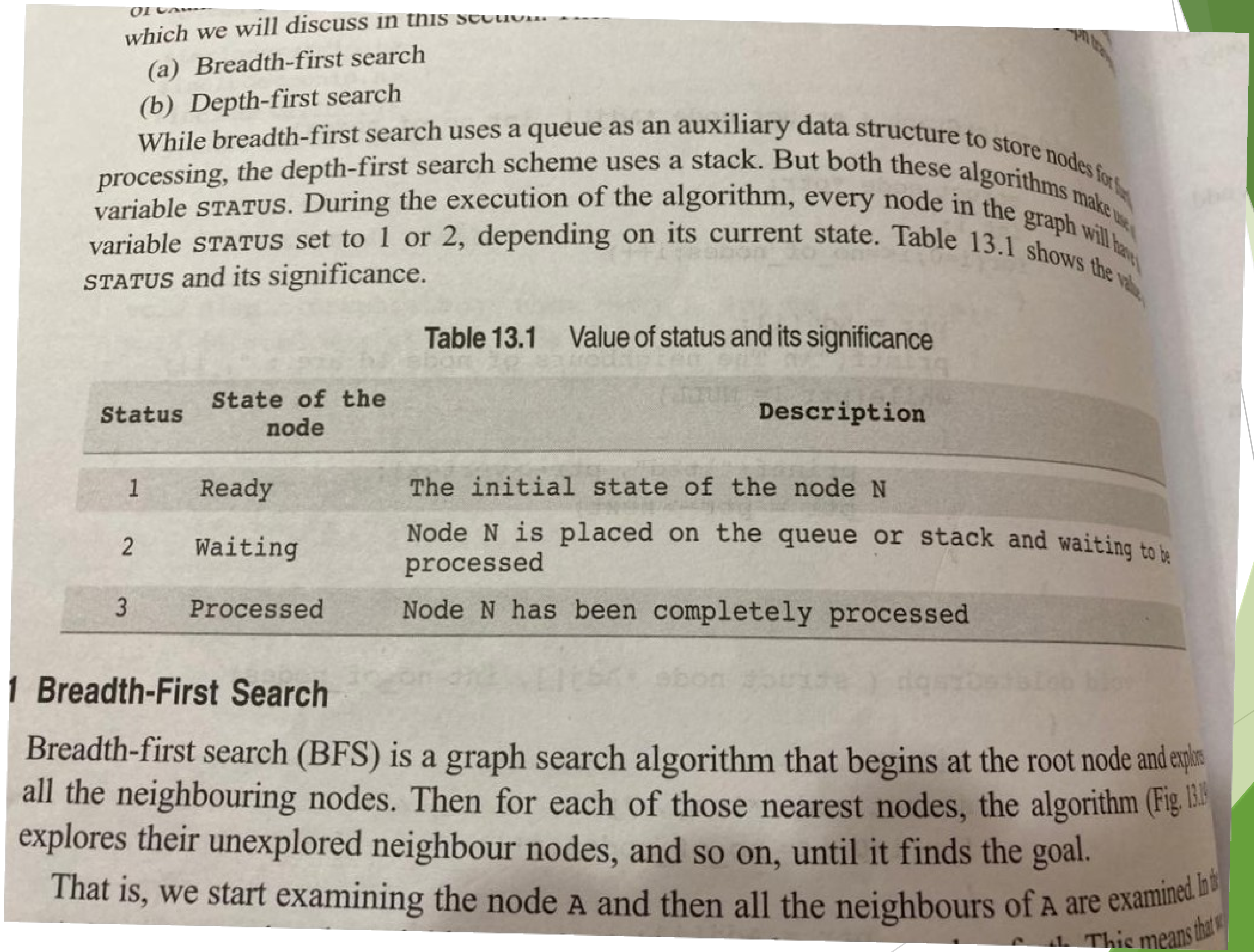
Breadth First Search

Depth First Search

Breadth First Search (BFS) Algorithm

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

Use Queue



which we will discuss in this section.

(a) Breadth-first search

(b) Depth-first search

While breadth-first search uses a queue as an auxiliary data structure to store nodes for fu processing, the depth-first search scheme uses a stack. But both these algorithms make use variable STATUS. During the execution of the algorithm, every node in the graph will have variable STATUS set to 1 or 2, depending on its current state. Table 13.1 shows the val STATUS and its significance.

Table 13.1 Value of status and its significance

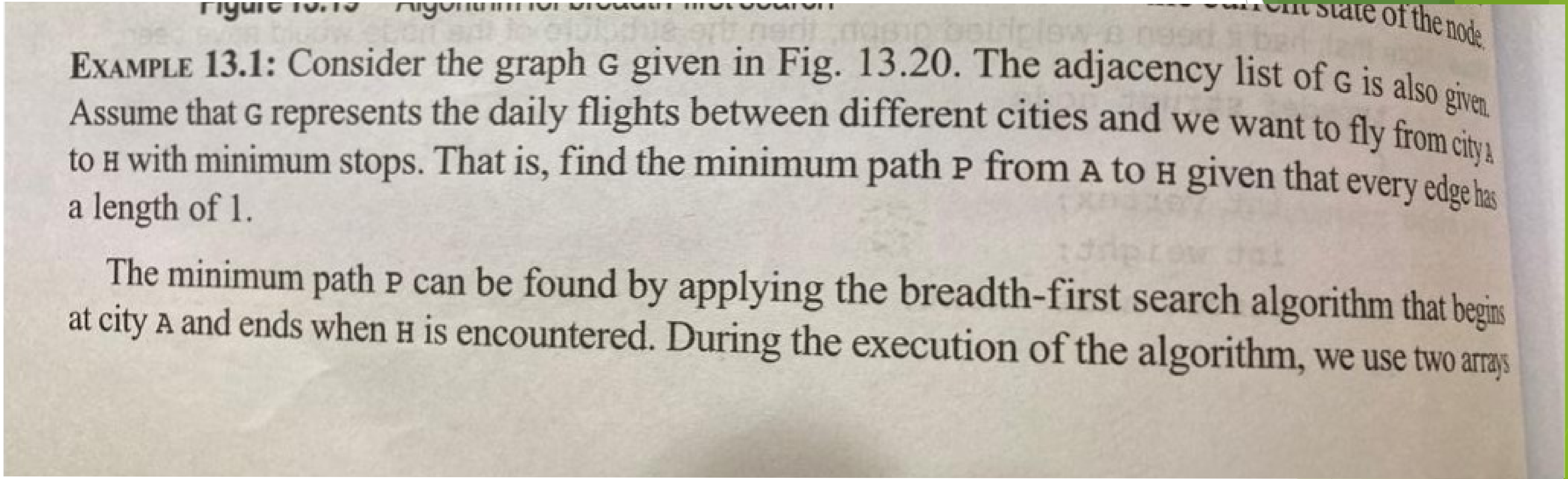
Status	State of the node	Description
1	Ready	The initial state of the node N
2	Waiting	Node N is placed on the queue or stack and waiting to be processed
3	Processed	Node N has been completely processed

1 Breadth-First Search

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm (Fig. 13.1 explores their unexplored neighbour nodes, and so on, until it finds the goal.

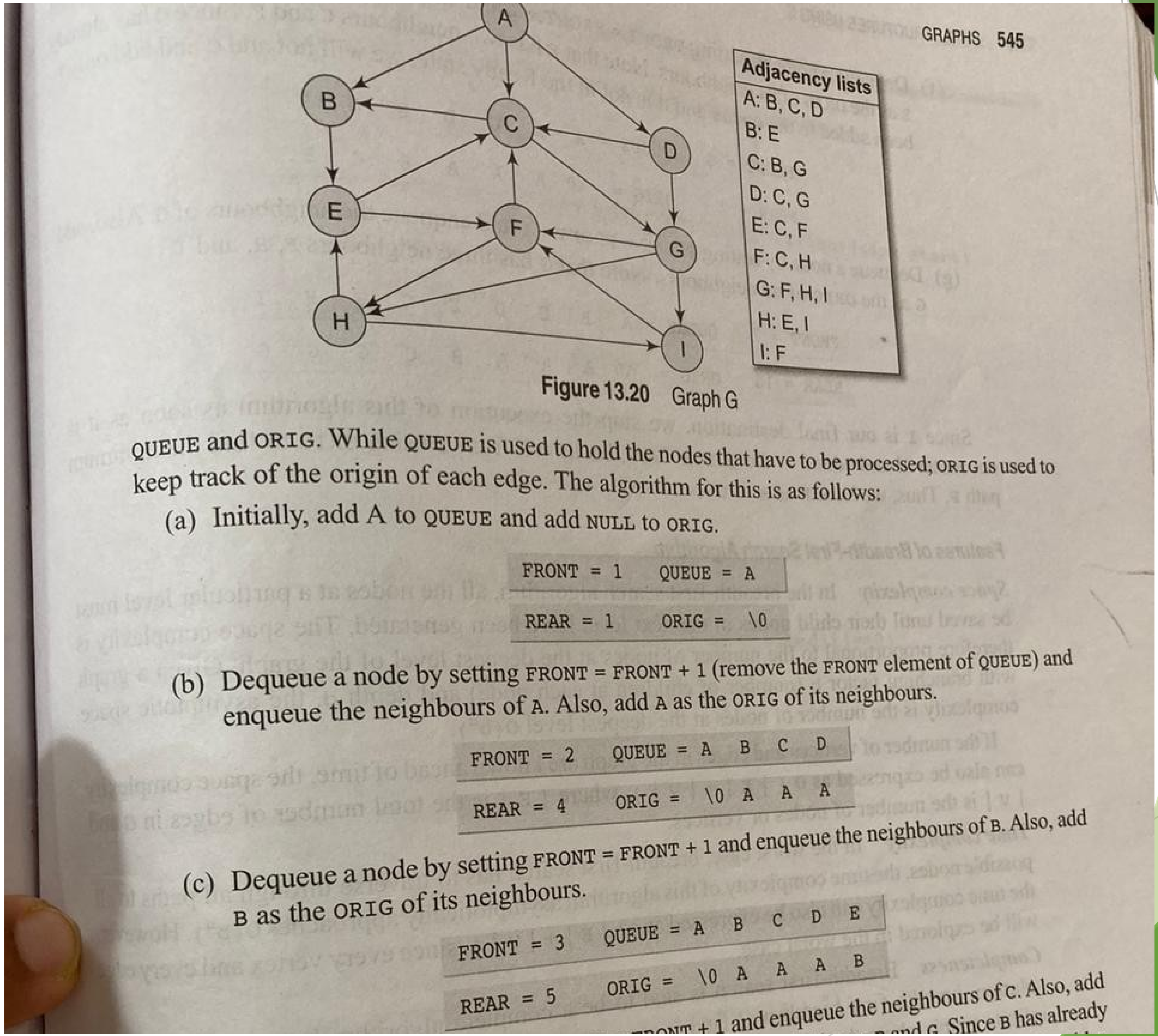
That is, we start examining the node A and then all the neighbours of A are examined. In th

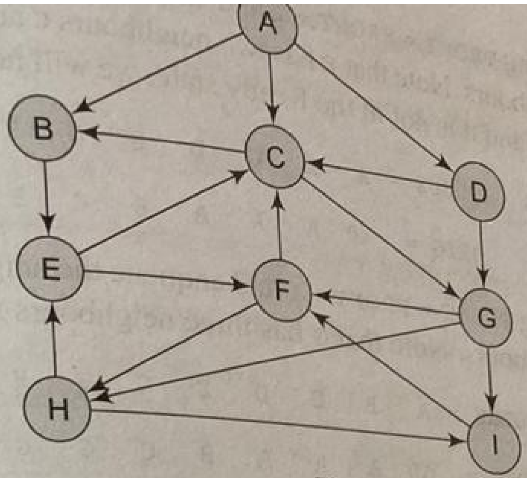
This means that w


state of the node.

Example 13.1: Consider the graph G given in Fig. 13.20. The adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to H with minimum stops. That is, find the minimum path P from A to H given that every edge has a length of 1.

The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when H is encountered. During the execution of the algorithm, we use two arrays





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

Figure 13.20 Graph G

QUEUE and ORIG. While QUEUE is used to hold the nodes that have to be processed; ORIG is used to keep track of the origin of each edge. The algorithm for this is as follows:

(a) Initially, add A to QUEUE and add NULL to ORIG.

```
FRONT = 1    QUEUE = A
REAR = 1     ORIG = \0
```

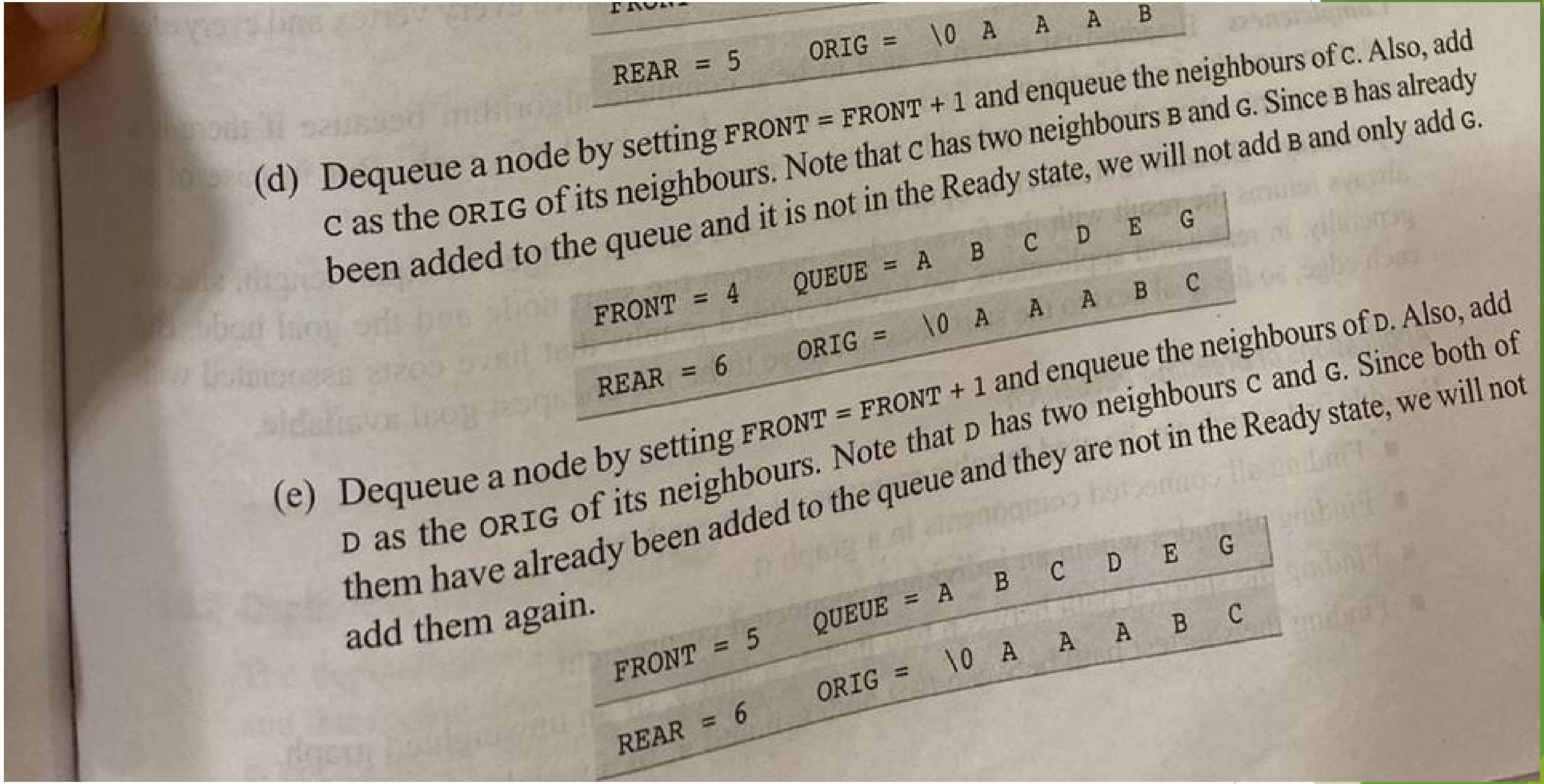
(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

```
FRONT = 2    QUEUE = A  B  C  D
REAR = 4     ORIG = \0  A  A  A
```

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

```
FRONT = 3    QUEUE = A  B  C  D  E
REAR = 5     ORIG = \0  A  A  A  B
```

FRONT + 1 and enqueue the neighbours of C. Also, add
and G. Since B has already



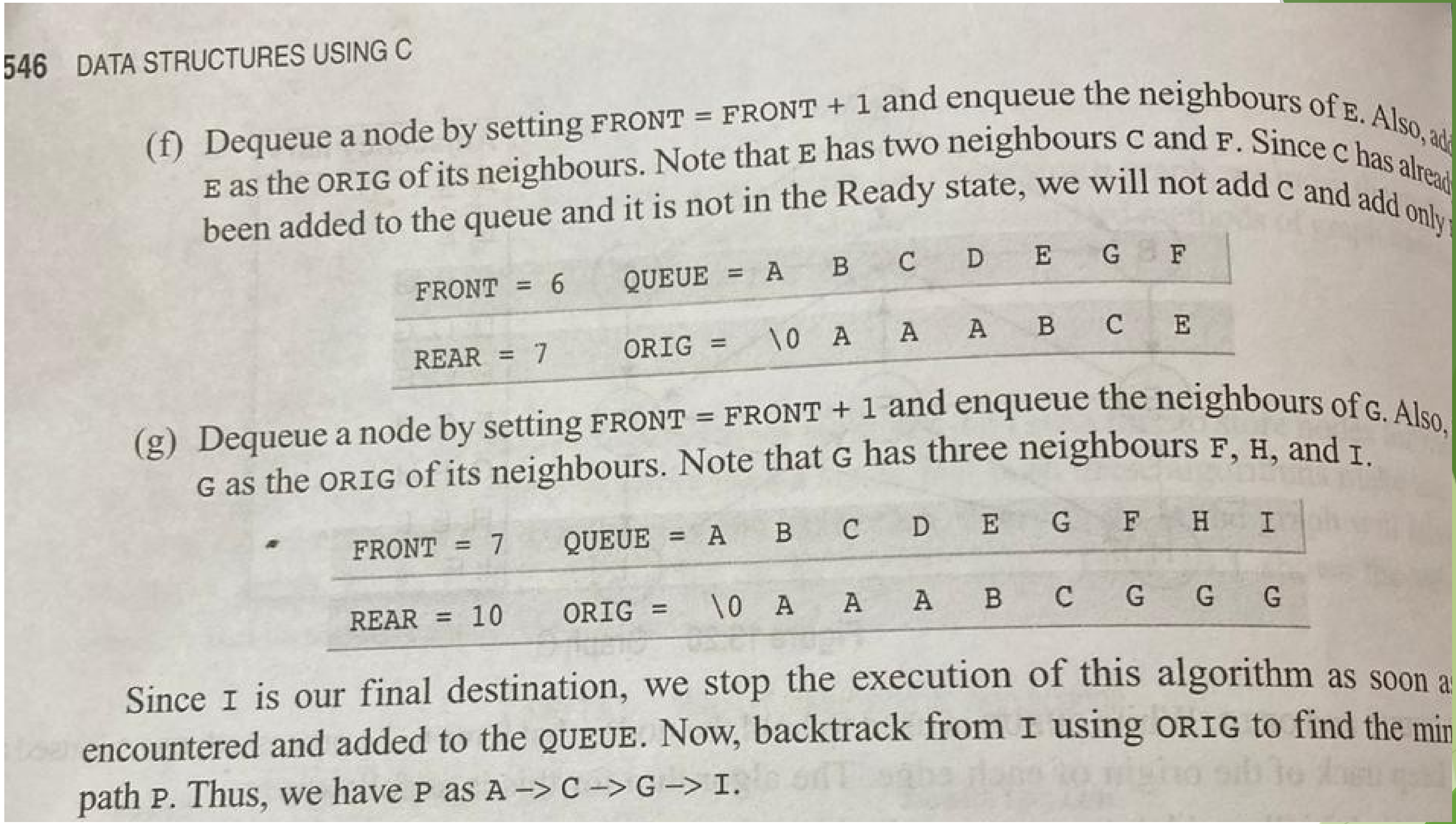
```
REAR = 5      ORIG = \0  A  A  A  B
```

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

```
FRONT = 4     QUEUE = A  B  C  D  E  G
REAR = 6      ORIG = \0  A  A  A  B  C
```

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

```
FRONT = 5     QUEUE = A  B  C  D  E  G
REAR = 6      ORIG = \0  A  A  A  B  C
```



(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, ad
E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has alread
been added to the queue and it is not in the Ready state, we will not add C and add only

FRONT = 6	QUEUE =	A	B	C	D	E	G	F	
REAR = 7	ORIG =	\0	A	A	A	B	C	E	

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also,
G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

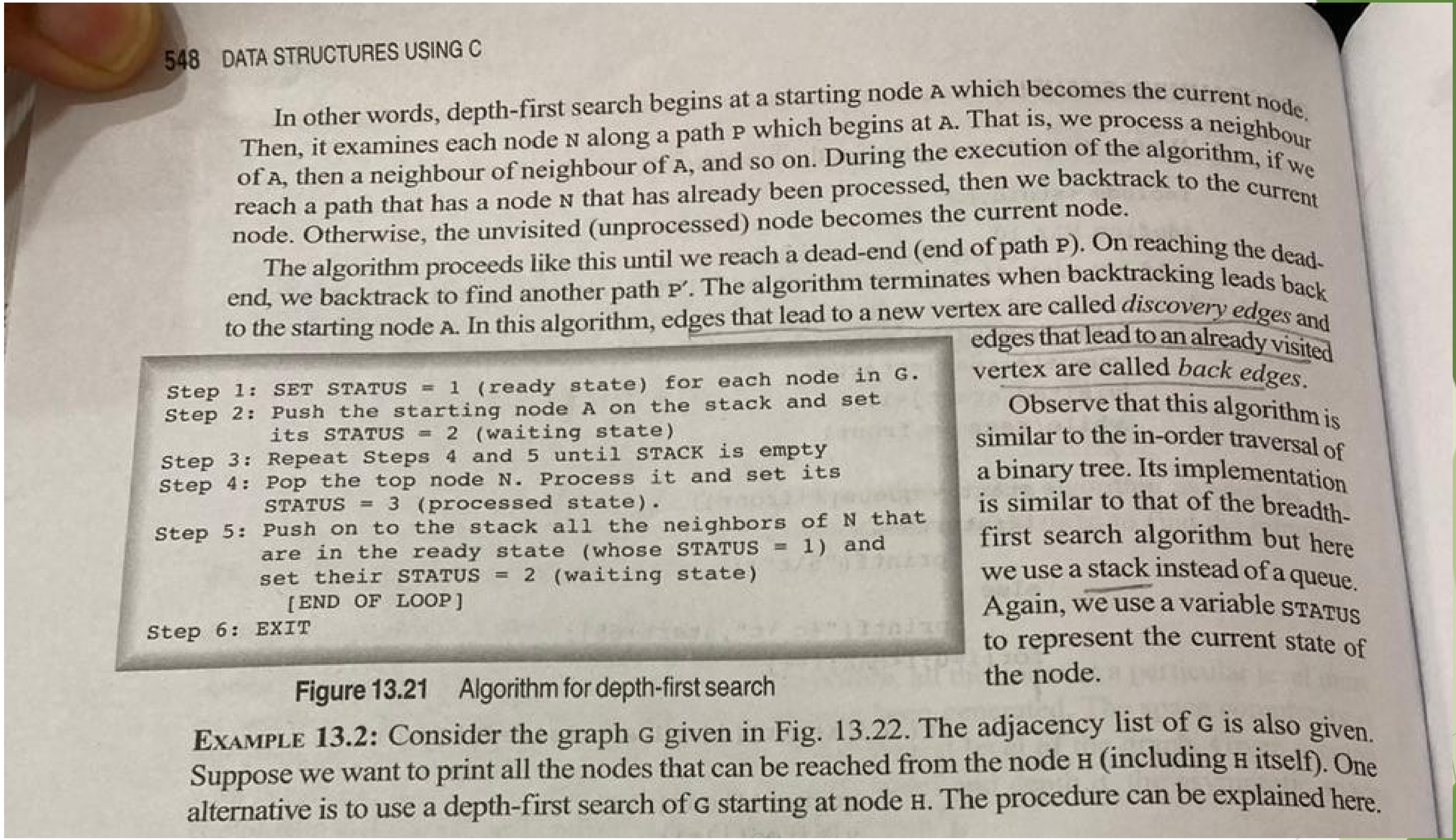
FRONT = 7	QUEUE =	A	B	C	D	E	G	F	H	I
REAR = 10	ORIG =	\0	A	A	A	B	C	G	G	G

Since I is our final destination, we stop the execution of this algorithm as soon a
encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the mi
path P. Thus, we have P as A -> C -> G -> I.

Depth First Search (DFS) Algorithm

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Use Stack



In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.

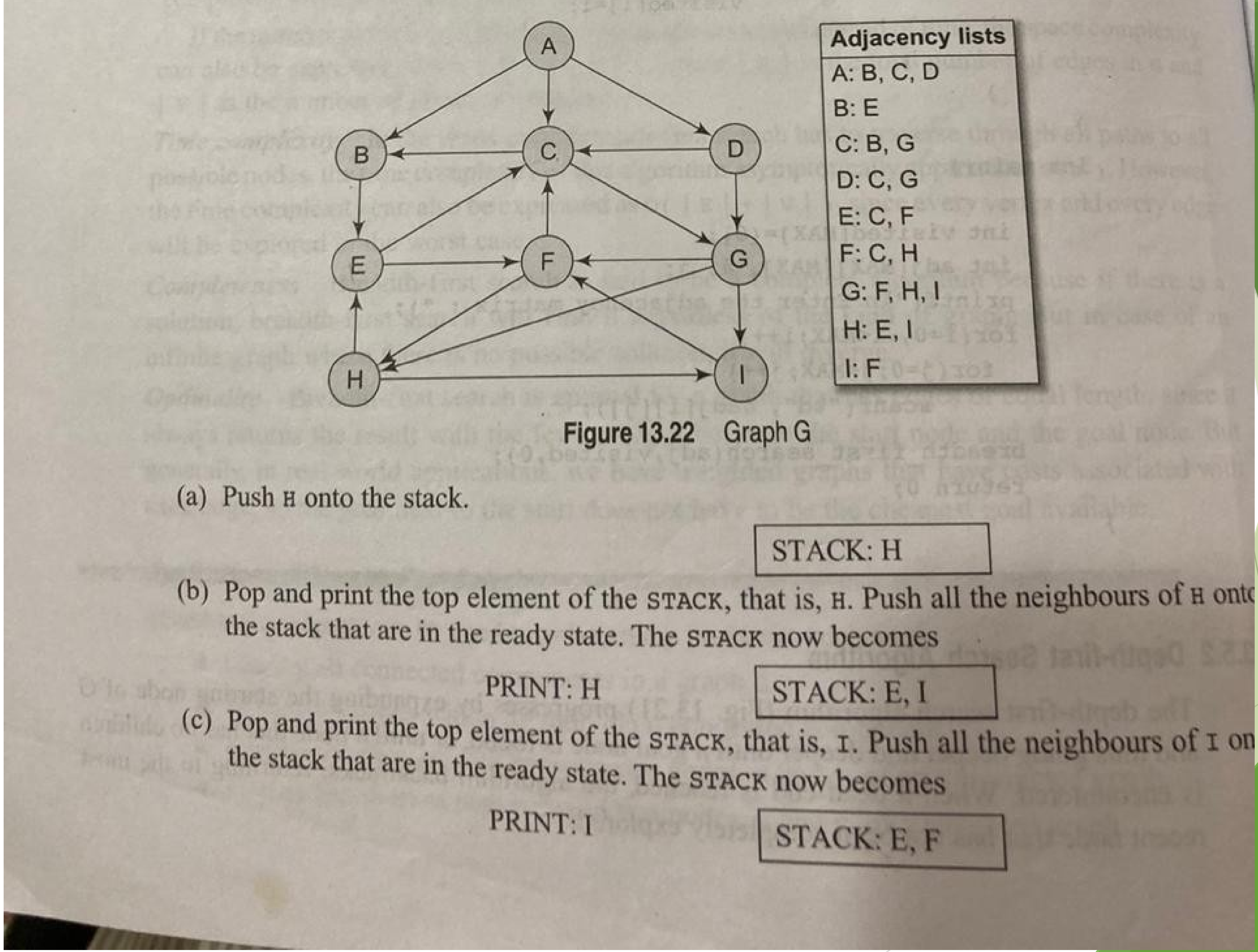
The algorithm proceeds like this until we reach a dead-end (end of path P). On reaching the dead-end, we backtrack to find another path P′. The algorithm terminates when backtracking leads back to the starting node A. In this algorithm, edges that lead to a new vertex are called *discovery edges* and edges that lead to an already visited vertex are called *back edges*.

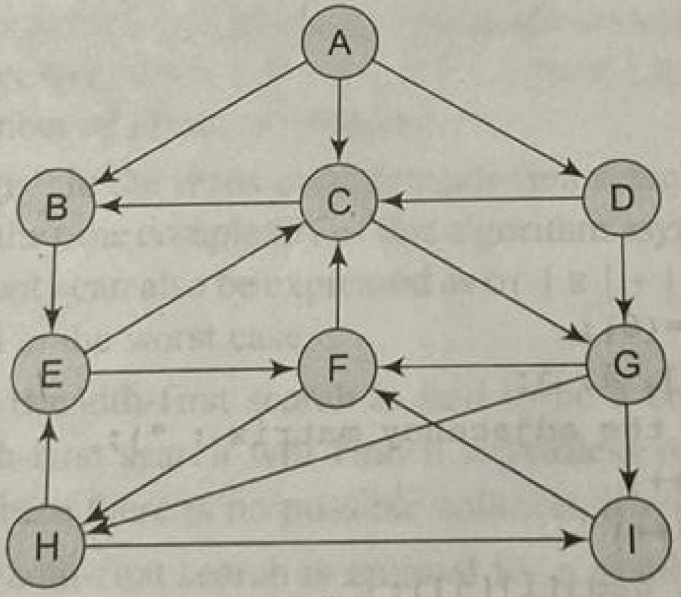
```
Step 1: SET STATUS = 1 (ready state) for each node in G.
Step 2: Push the starting node A on the stack and set
        its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4: Pop the top node N. Process it and set its
        STATUS = 3 (processed state).
Step 5: Push on to the stack all the neighbors of N that
        are in the ready state (whose STATUS = 1) and
        set their STATUS = 2 (waiting state)
          [END OF LOOP]
Step 6: EXIT
```

Figure 13.21 Algorithm for depth-first search

Observe that this algorithm is similar to the in-order traversal of a binary tree. Its implementation is similar to that of the breadth-first search algorithm but here we use a stack instead of a queue. Again, we use a variable STATUS to represent the current state of the node.

Example 13.2: Consider the graph G given in Fig. 13.22. The adjacency list of G is also given. Suppose we want to print all the nodes that can be reached from the node H (including H itself). One alternative is to use a depth-first search of G starting at node H. The procedure can be explained here.





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

Figure 13.22 Graph G

(a) Push H onto the stack.

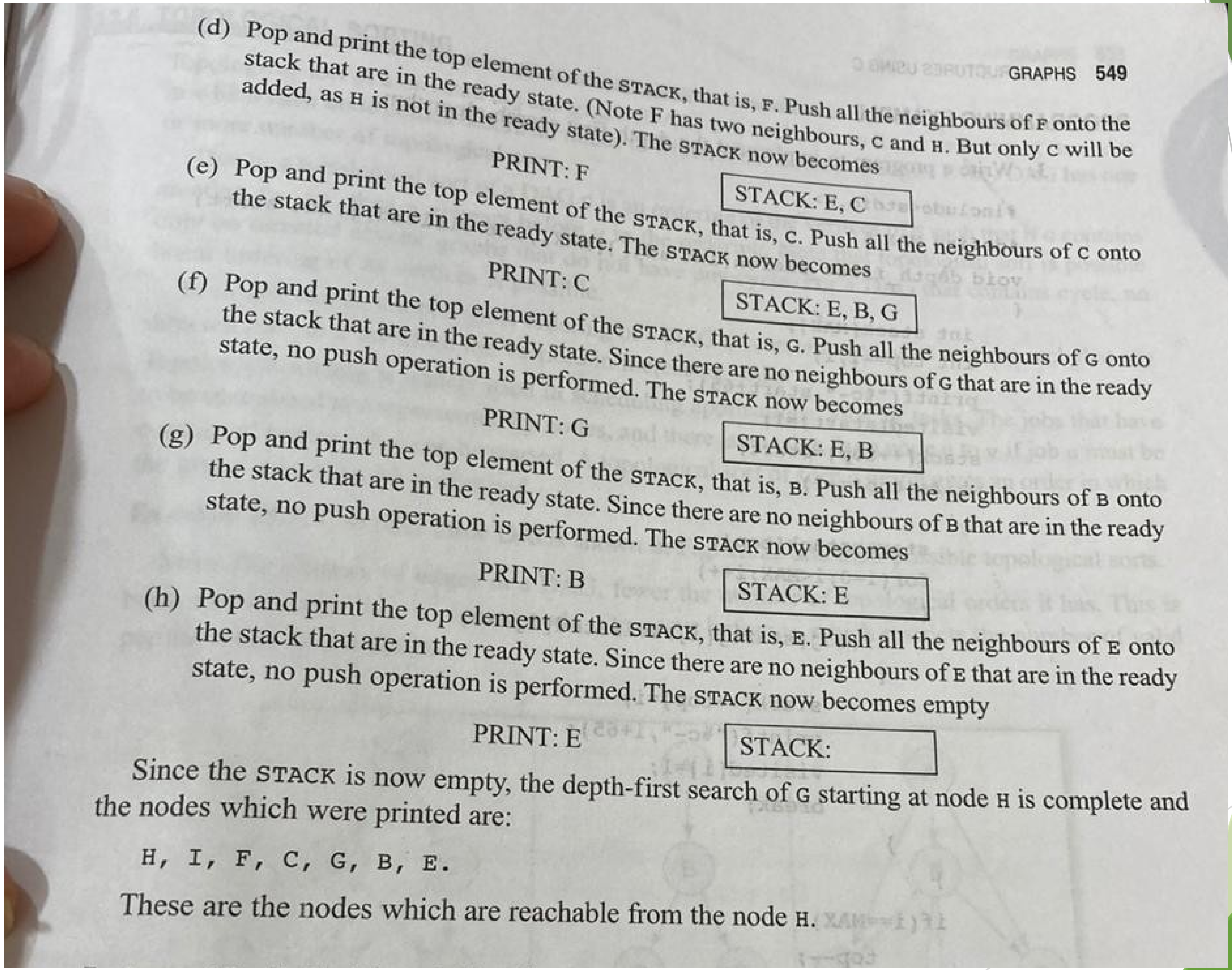
STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H ont
the stack that are in the ready state. The STACK now becomes

PRINT: H STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I on
the stack that are in the ready state. The STACK now becomes

PRINT: I STACK: E, F



(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state). The STACK now becomes

PRINT: F STACK: E, C

(e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C STACK: E, B, G

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G STACK: E, B

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B STACK: E

(h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty

PRINT: E STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

```
H, I, F, C, G, B, E.
```

These are the nodes which are reachable from the node H.

Shortest path

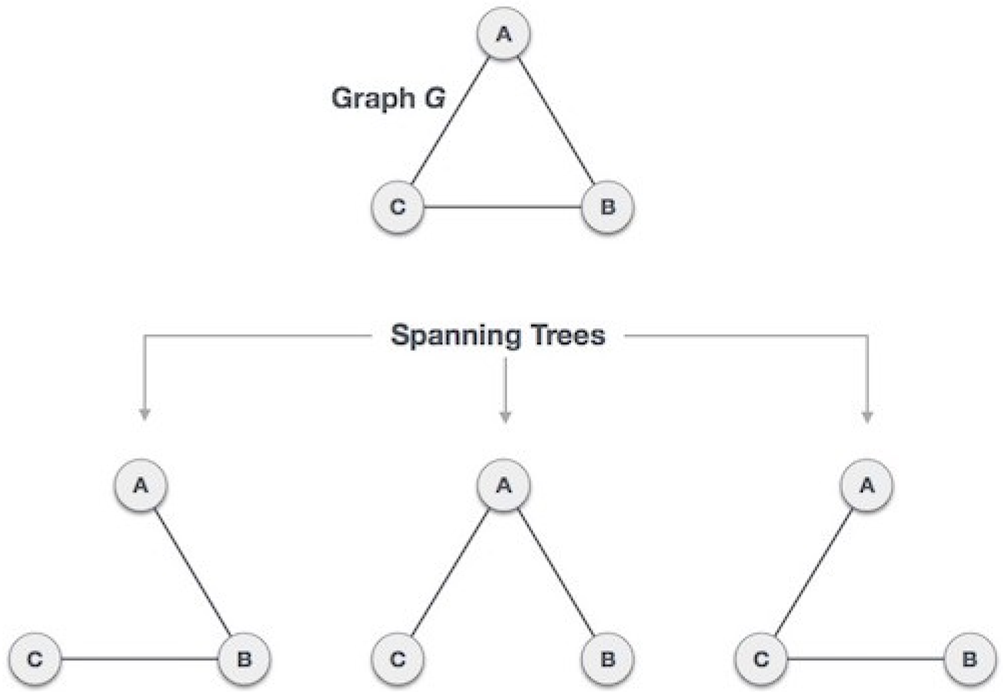
Spanning tree and Minimal spanning tree

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible.

Shortest path

spanning tree



Shortest path

spanning tree

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

Civil Network Planning

Computer Network Routing Protocol

Cluster Analysis

Shortest path

spanning tree

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

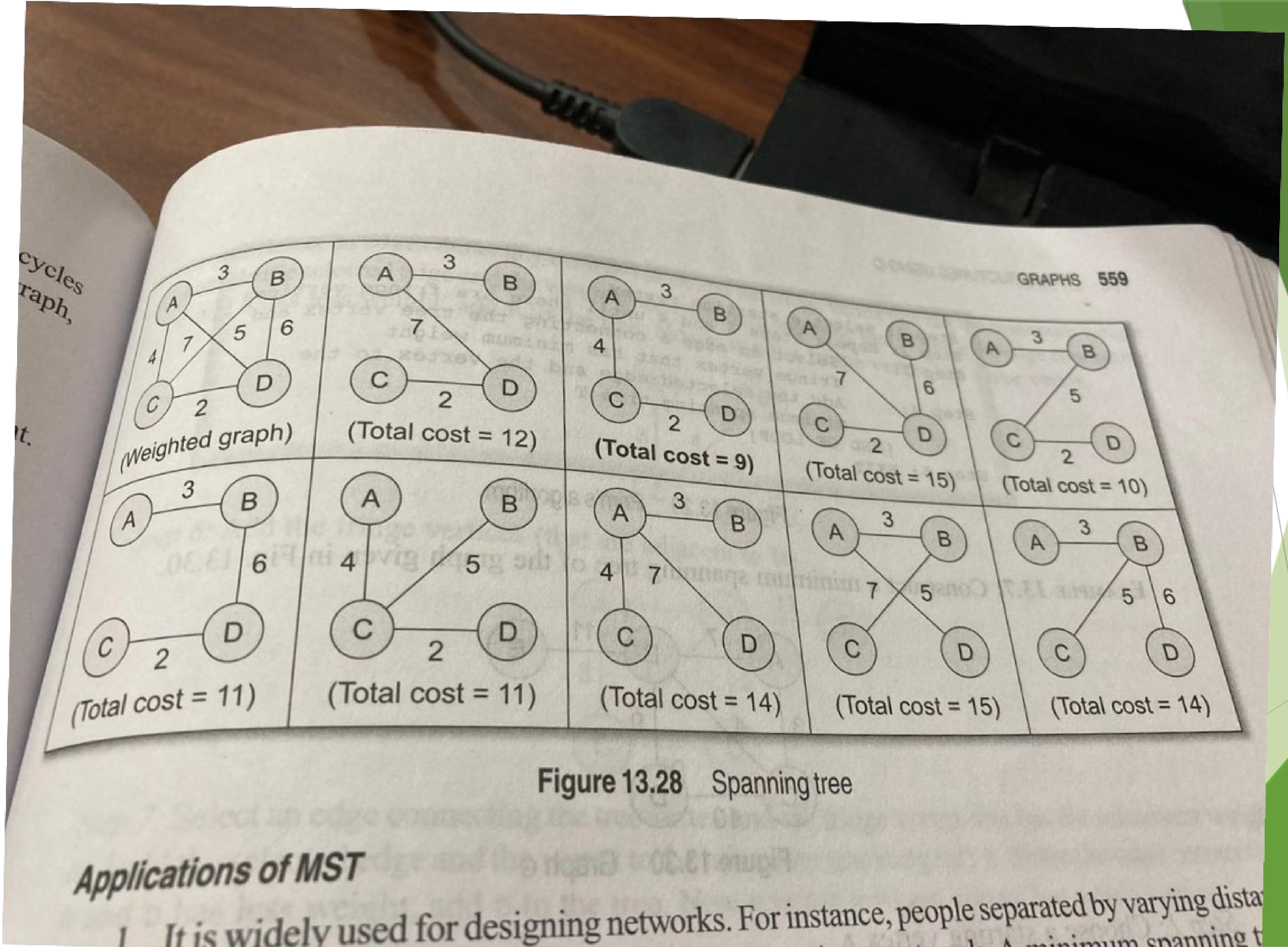
A connected graph G can have more than one spanning tree.

All possible spanning trees of graph G, have the same number of edges and vertices.

The spanning tree does not have any cycle (loops).

Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.

Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.



cycles
raph,

t.

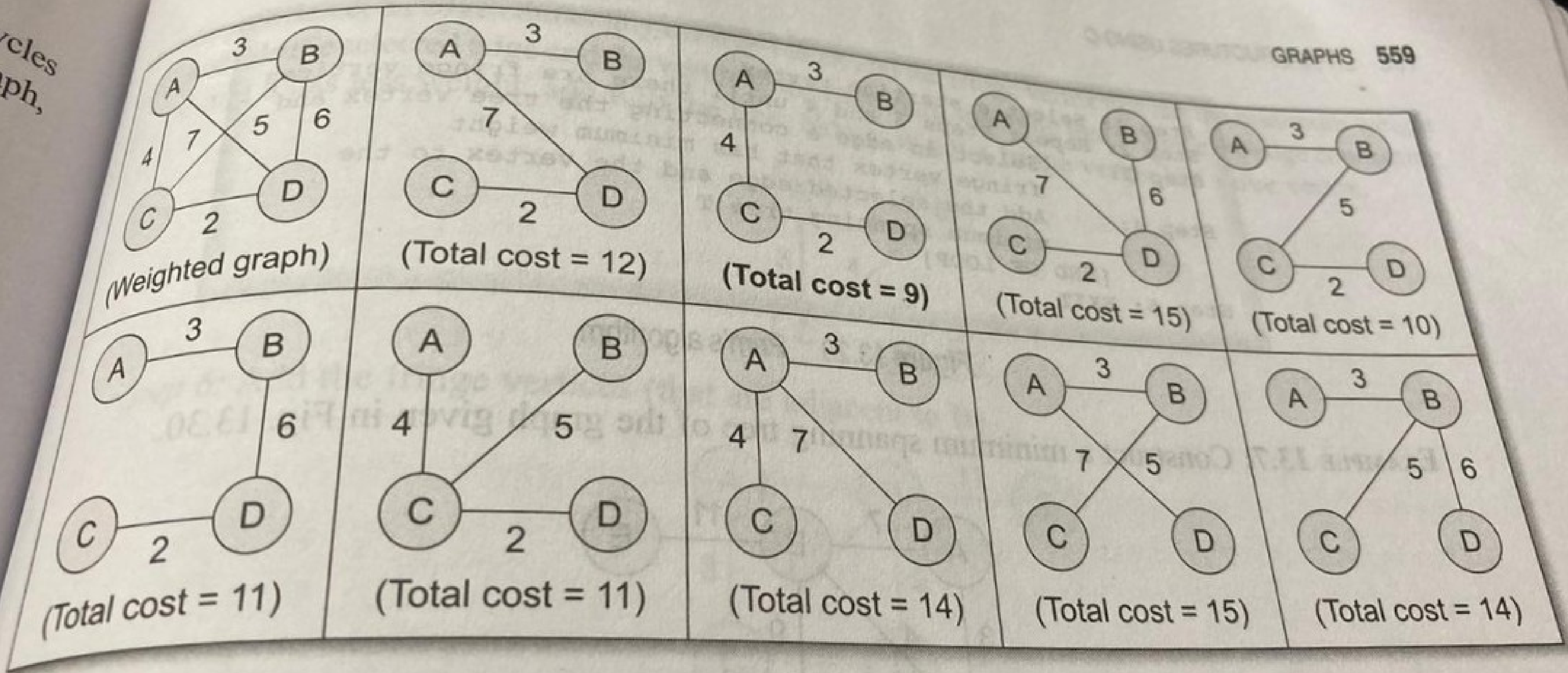


Figure 13.28 Spanning tree

Applications of MST

1. It is widely used for designing networks. For instance, people separated by varying dista

minimum spanning t

Shortest path

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

Kruskal's Algorithm

Prim's Algorithm

Both are greedy algorithms.