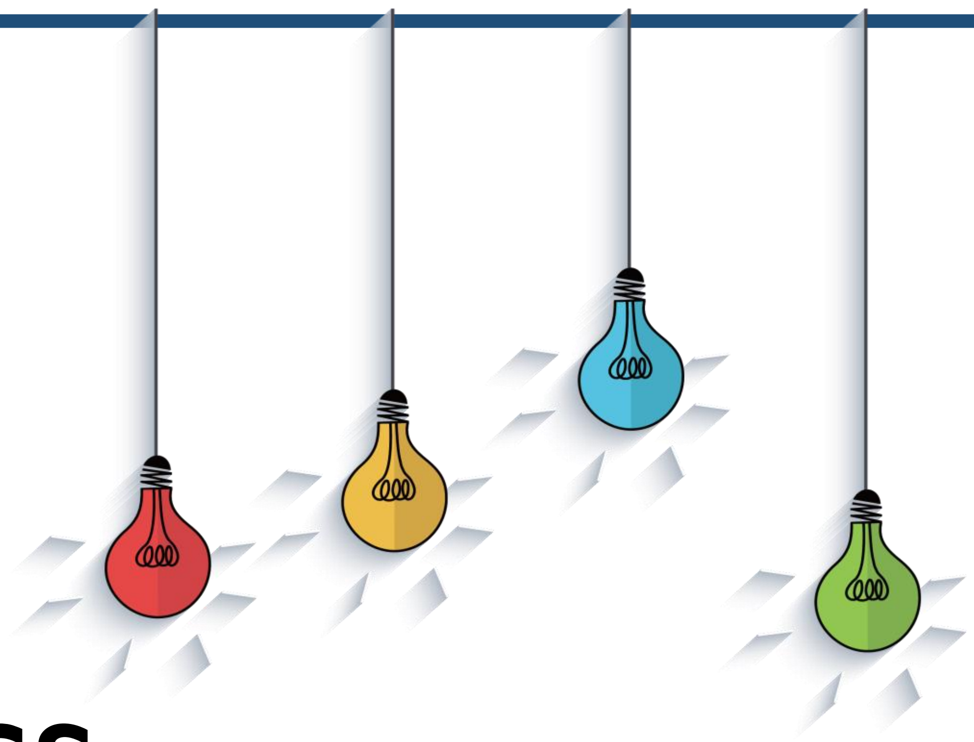
IF120 Discrete Mathematics

12 Trees 1

Angga Aditya Permana, Januar Wahjudi, Yaya Suryana, Meriske, Muhammad Fahrury Romdendine



REVIEW

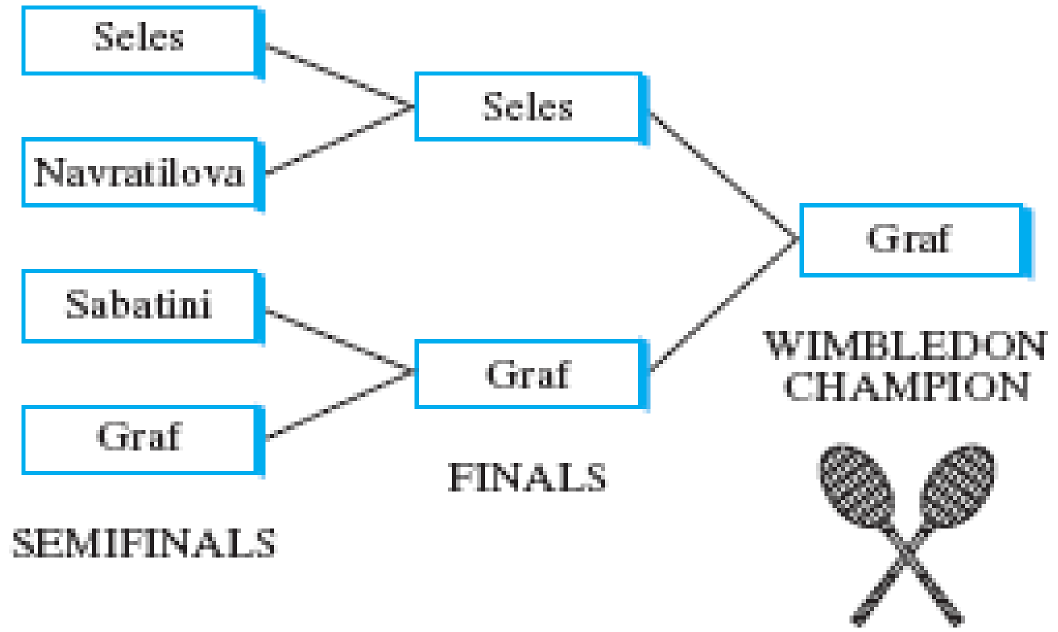
- Graph Representation
- Isomorphism
- Planar Graphs
- Shortest Path Problem

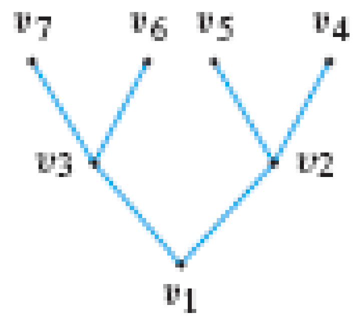
OUTLINE

- Trees' Terminologies
- Spanning Trees
- Binary Trees

Trees

- Figure below shows the results of the semifinals and finals of a classic tennis competition at Wimbledon, which featured four of the greatest players in the history of tennis. At Wimbledon, when a player loses, she is out of the tournament. Winners continue to play until only one person, the champion, remains. (Such a competition is called a *single elimination tournament.*)





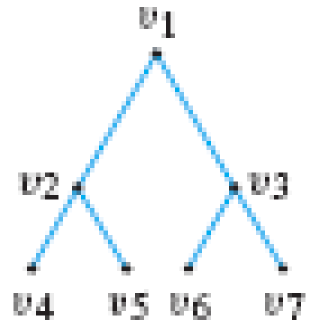
Trees

- **Definition 12.1**: A (*free*) **tree** *T* is a simple graph satisfying the following: If *v* and *w* are vertices in *T*, there is a unique simple path from *v* to *w*.
- A **rooted tree** is a tree in which a particular vertex is designated the root.

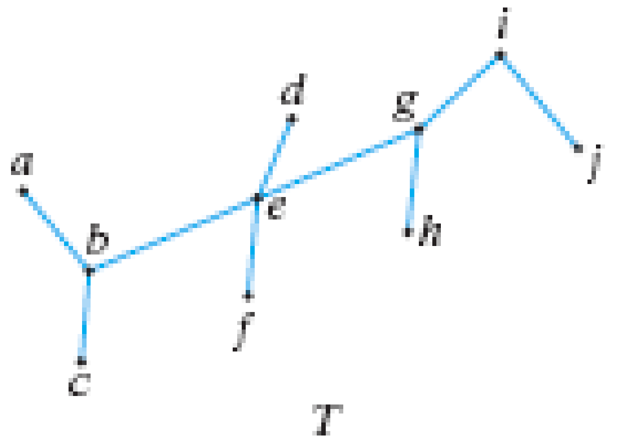
❑ If we designate the winner as the root, the single-elimination tournament is a rooted tree. Notice that if *v* and *w* are vertices in this graph, there is a unique simple path from *v* to *w*. For example, the unique simple path from v_2 to v_7 is (v_2, v_1, v_3, v_7).

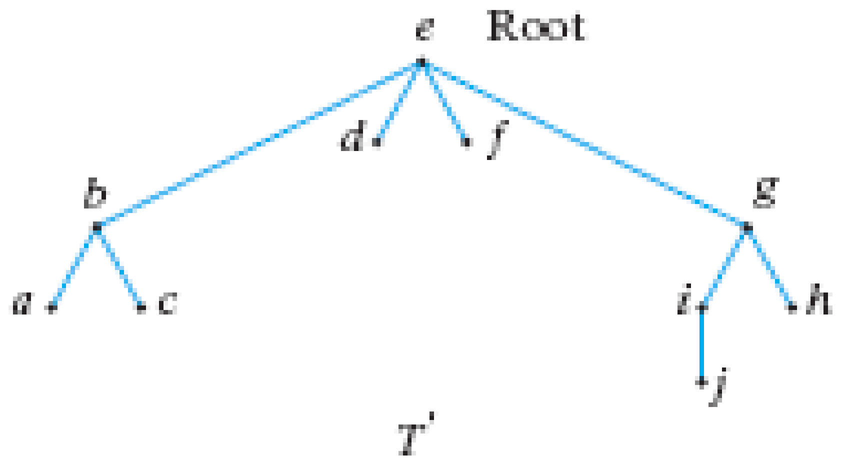
- In contrast to natural trees, which have their roots at the bottom, in graph theory rooted trees are typically drawn with their roots at the top. We call the level of the root level 0. The vertices under the root are said to be on level 1, and so on. Thus the **level of a vertex** *v* is the length of the simple path from the root to *v*. The **height** of a rooted tree is the maximum level number that occurs.

Examples

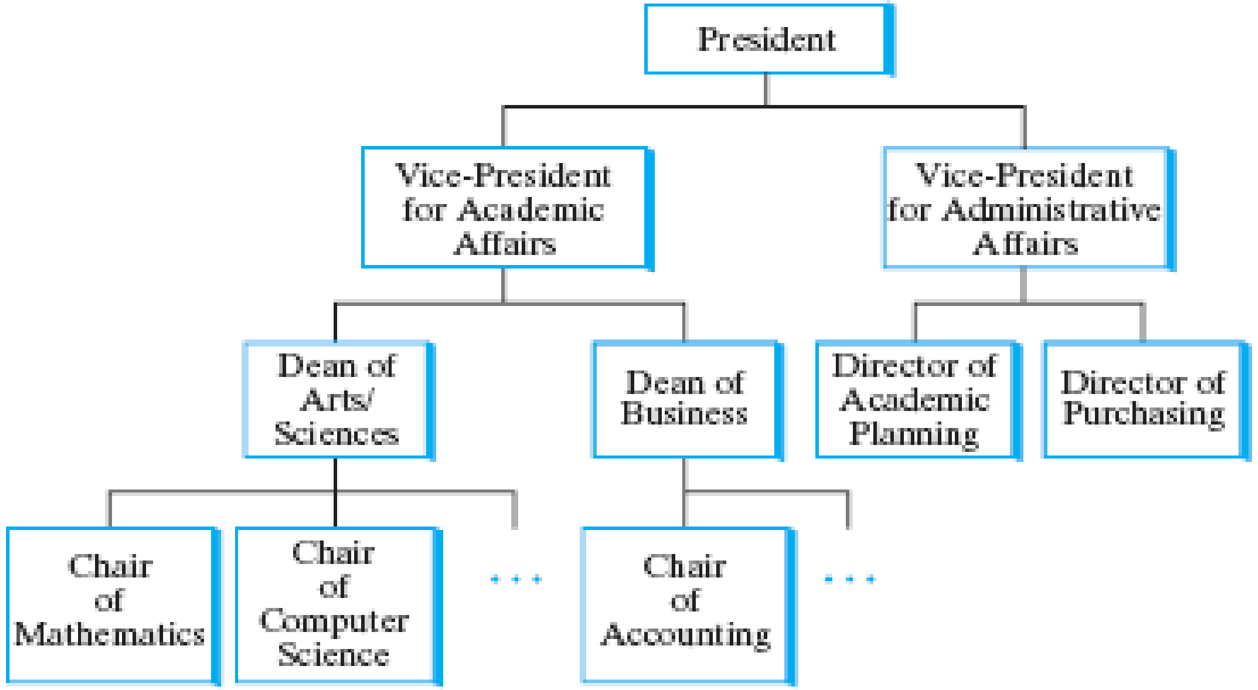


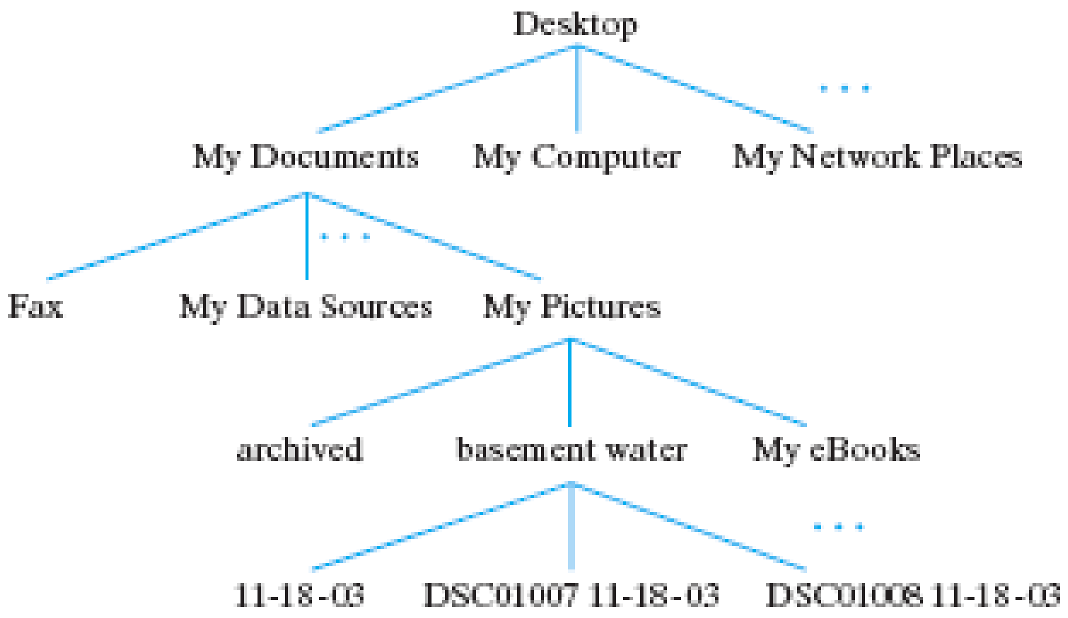
- The vertices $v_1, v_2, v_3, v_4, v_5, v_6, v_7$ in the rooted tree of Figure on the right are on (respectively) levels 0, 1, 1, 2, 2, 2, 2. The height of the tree is 2.
- If we designate e as the root in the tree *T* of Figure below, we obtain the rooted tree *T'* shown. The vertices *a*, *b*, *c*, *d*, e, *f*, *g*, *h*, *i*, *j* are on (respectively) levels 2, 1, 2, 1, 0, 1, 1, 2, 2, 3. The height of *T'* is 3.

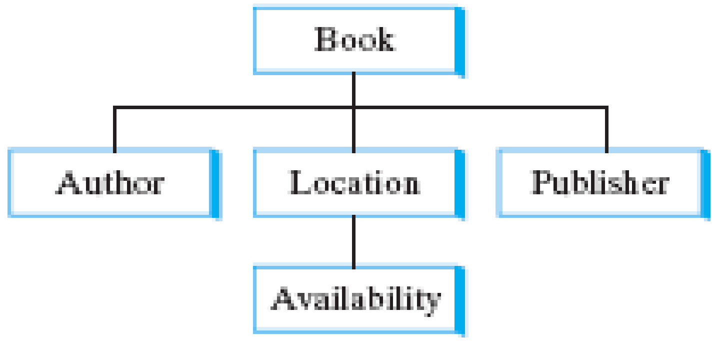




Examples

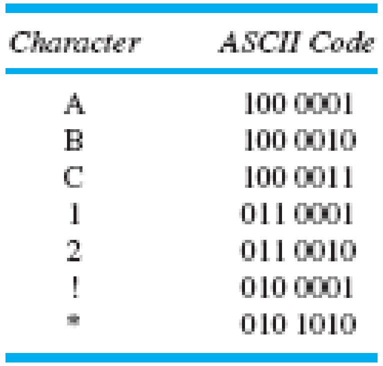

President
Vice-President for Academic Affairs
Vice-President for Administrative Affairs
Dean of Arts/ Sciences
Dean of Business
Director of Academic Planning
Director of Purchasing
Chair of Mathematics
Chair of Computer Science
. . .
Chair of Accounting
. . .


Desktop
My Documents
My Computer
My Network Places
. . .
Fax
My Data Sources
My Pictures
. . .
archived
basement water
My eBooks
11-18-03
DSC01007 11-18-03
DSC01008 11-18-03
. . .


Book
Author
Location
Publisher
Availability

Huffman Codes

- The most common way to represent characters internally in a computer is by using fixed-length bit strings. For example, **ASCII** (American Standard Code for Information Interchange) represents each character by a string of seven bits. Examples are given in Table below.



Character	*ASCII Code*
A	100 0001
B	100 0010
C	100 0011
1	011 0001
2	011 0010
!	010 0001
*	010 1010

- **Huffman codes,** which represent characters by variable-length bit strings, provide alternatives to ASCII and other fixed-length codes. The idea is to use short bit strings to represent the most frequently used characters and to use longer bit strings to represent less frequently used characters. In this way it is generally possible to represent strings of characters, such as text and programs, in less space than if ASCII were used.

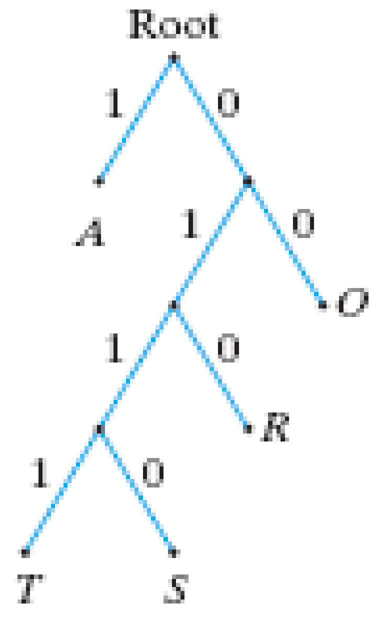
Constructing an Optimal Huffman Code

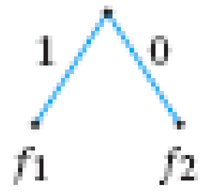
This algorithm constructs an optimal Huffman code from a table giving the frequency of occurrence of the characters to be represented. The output is a rooted tree with the vertices at the lowest levels labeled with the frequencies and with the edges labeled with bits as in Figure 9.1.10. The coding tree is obtained by replacing each frequency with a character having that frequency.

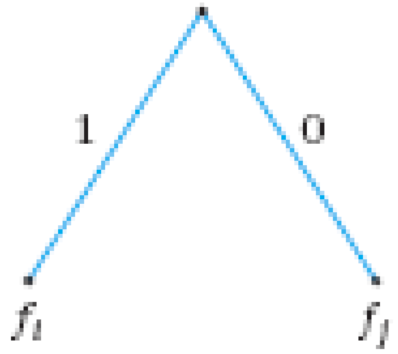
Input: A sequence of n frequencies, $n \geq 2$

Output: A rooted tree that defines an optimal Huffman code

```
huffman(f, n) {
    if (n == 2) {
        let $f_1$ and $f_2$ denote the frequencies
        let $T$ be as in Figure 9.1.11
        return $T$
    }
    let $f_i$ and $f_j$ denote the smallest frequencies
    replace $f_i$ and $f_j$ in the list $f$ by $f_i + f_j$
    $T' = huffman(f, n - 1)$
    replace a vertex in $T'$ labeled $f_i + f_j$ by the tree shown in Figure 9.1.12
        to obtain the tree $T$
    return $T$
}
```







Huffman Codes

- We show how Algorithm 12.1 constructs an optimal Huffman code using Table below.

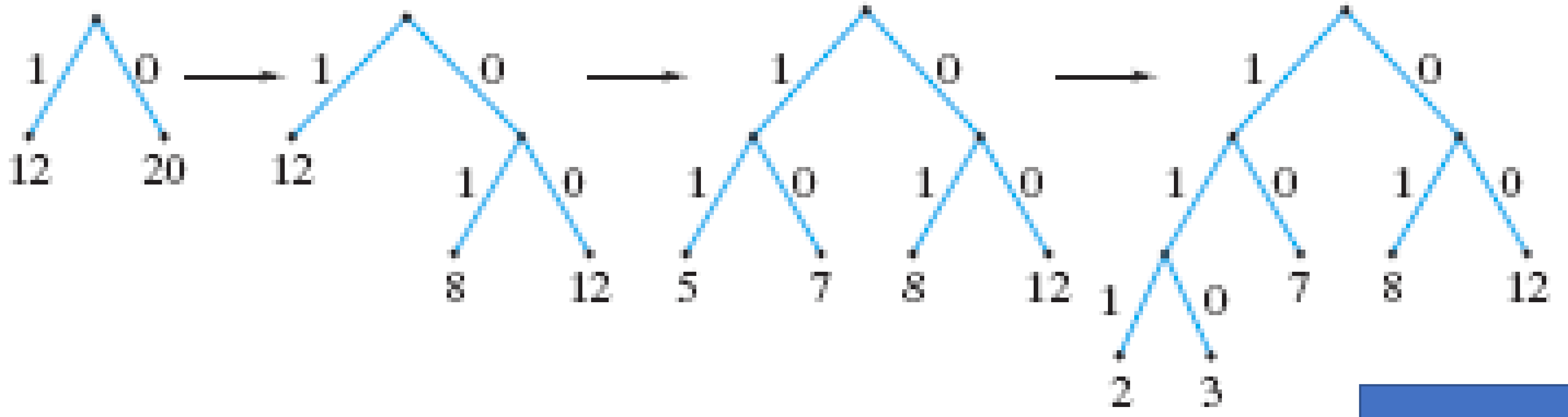
Character	*Frequency*
!	2
@	3
#	7
$	8
%	12

The algorithm begins by repeatedly replacing the smallest two frequencies with the sum until a two-element sequence is obtained:

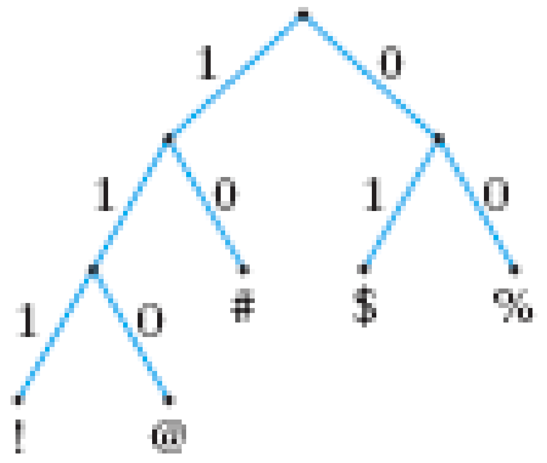
$$\begin{gathered} 2,3,7,8,12 \rightarrow 2+3,7,8,12 \\ 5,7,8,12 \rightarrow 5+7,8,12 \\ 8,12,12 \rightarrow 8+12,12 \\ 12,20 \end{gathered}$$

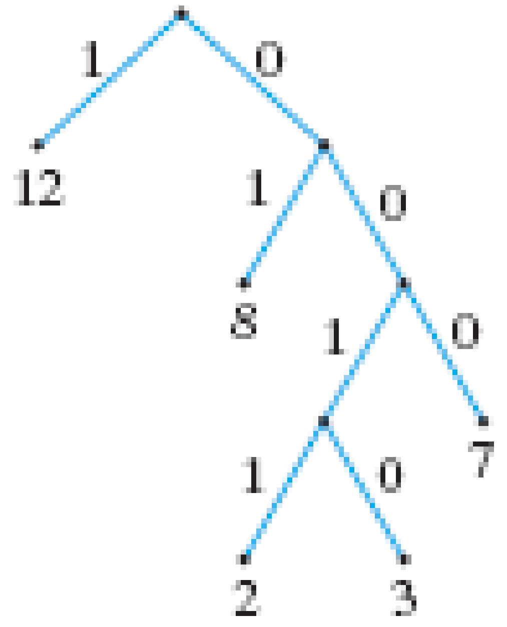
Huffman Codes

The algorithm then constructs trees working backward beginning with the two-element sequence 12, 20 as shown in Figure below. For example, the second tree is obtained from the first by replacing the vertex labeled 20 by 8 and 12. Finally, to obtain the optimal Huffman coding tree, we replace each frequency by a character having that frequency (see the figure on the next slide).



Huffman Codes





Notice that the Huffman tree for the problem is not unique. When 12 is replaced by 5, 7, because there are two vertices labeled 12, there is a choice. In the final tree of previous figure, we arbitrarily chose one of the vertices labeled 12. If we choose the other vertex labeled 12, we will obtain the tree of the top right figure. Either of the Huffman trees gives an optimal code; that is, either will encode text having the frequencies of the table in exactly the same (optimal) space.

Trees' Terminologies

- **Definition 12.2**: Let T be a tree with root v_0. Suppose that x, y, and z are vertices in T and that $(v_0, v_1, \ldots, v_n)$ is a simple path in T. Then

a) v_{n-1} is the **parent** of v_n.

b) $v_0, \ldots, v_{n-1}$ are **ancestors** of v_n.

c) v_n is a **child** of v_{n-1}.

d) If x is an ancestor of y, y is a **descendant** of x.

e) If x and y are children of z, x and y are **siblings**.

f) If x has no children, x is a **terminal vertex** (or a **leaf**).

g) If x is not a terminal vertex, x is an **internal** (or **branch**) vertex.

h) The **subtree of T rooted at x** is the graph with vertex set V and edge set E, where V is x together with the descendants of x and

$$E = \{e \mid e \text{ } is \text{ } an \text{ } edge \text{ } on \text{ } a \text{ } simple \text{ } path \text{ } from \text{ } x \text{ } to \text{ } some \text{ } vertex \text{ } in \text{ } V\}.$$

<u>Trees' Terminologies</u>

- A graph with no cycles is called an **acyclic graph.**
- A tree is a connected, acyclic graph.
- The converse is also true; every connected, acyclic graph is a tree.
- **<u>Theorem 12.1</u>**: *Let T be a graph with n vertices. The following are equivalent.*

a) *T is a tree.*

b) *T is connected and acyclic.*

c) *T is connected and has $n - 1$ edges.*

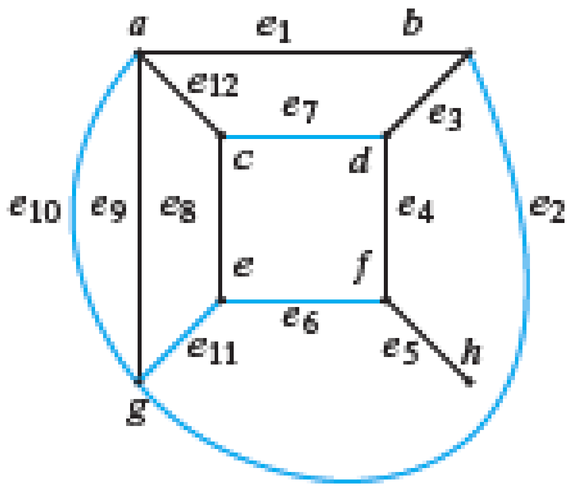
d) *T is acyclic and has $n - 1$ edges.*

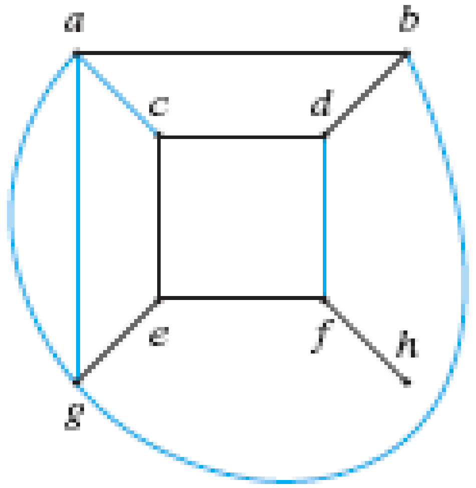
Spanning Trees

- **Definition 12.3**: A tree *T* is a **spanning tree** of a graph *G* if *T* is a subgraph of *G* that contains all of the vertices of *G*.

❑A spanning tree of the graph *G* of Figure below is shown in black.

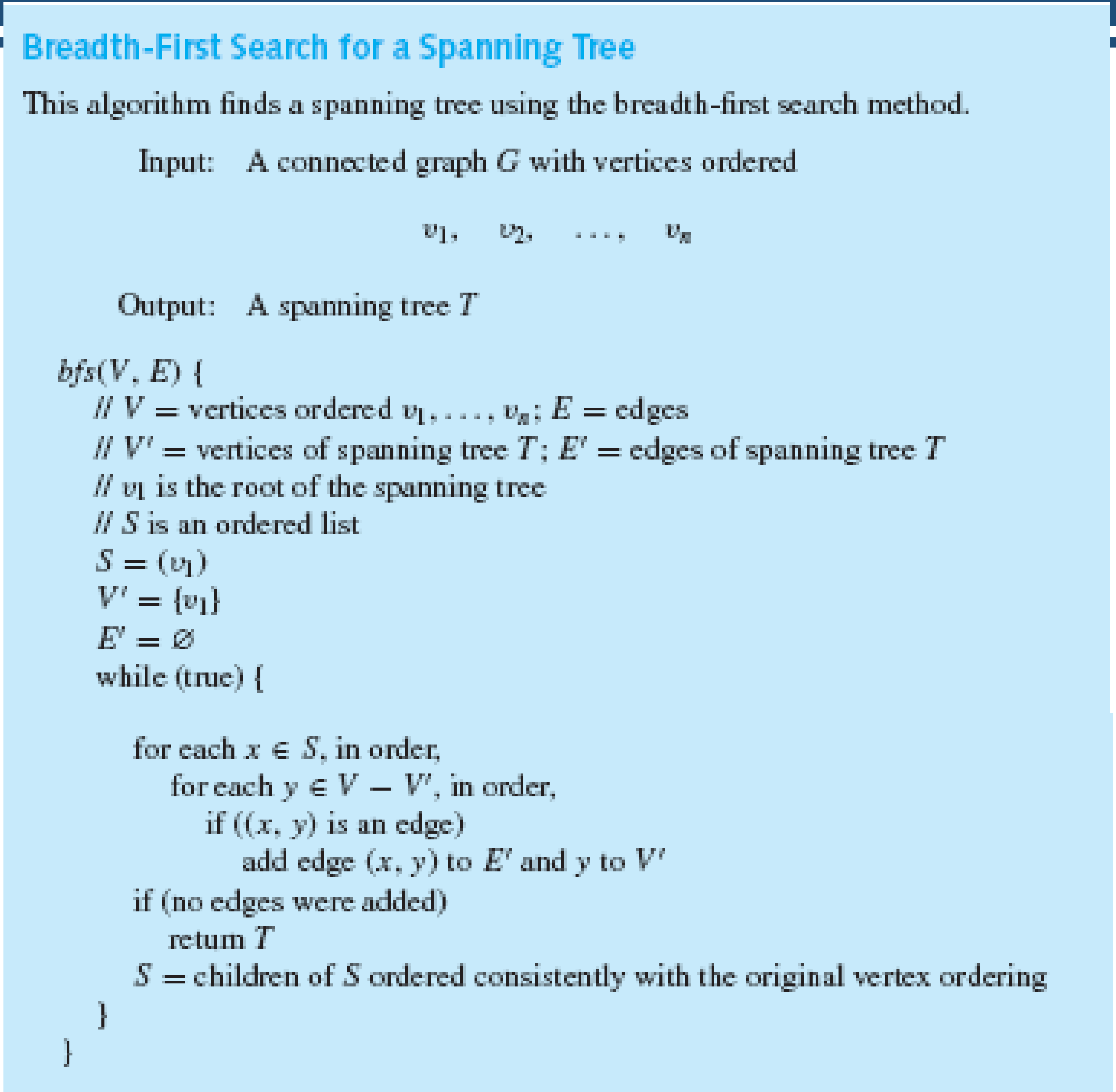
❑In general, a graph will have several spanning trees. Another spanning tree of the graph *G* is shown in Figure on the right (black color).





Spanning Trees

- **Theorem 12.2**: *A graph G has a spanning tree if and only if G is connected.*
- **Algorithm 12.2**: Breadth-First Search



Breadth-First Search for a Spanning Tree

This algorithm finds a spanning tree using the breadth-first search method.

Input: A connected graph G with vertices ordered

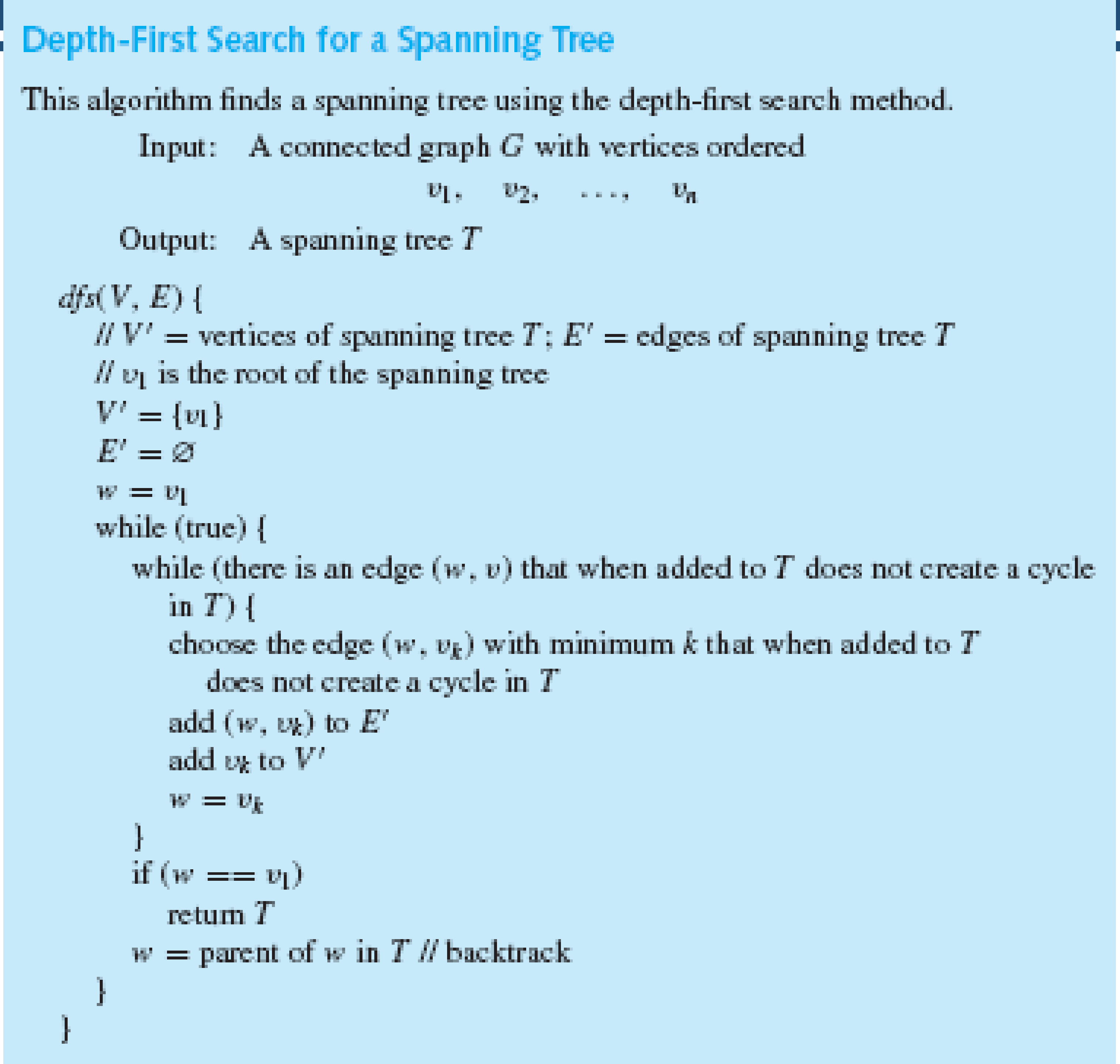
$$v_1, \quad v_2, \quad \ldots, \quad v_n$$

Output: A spanning tree T

```
bfs(V, E) {
    // $V$ = vertices ordered $v_1, \ldots, v_n$; $E$ = edges
    // $V'$ = vertices of spanning tree $T$; $E'$ = edges of spanning tree $T$
    // $v_1$ is the root of the spanning tree
    // $S$ is an ordered list
    $S = (v_1)$
    $V' = \{v_1\}$
    $E' = \varnothing$
    while (true) {
        for each $x \in S$, in order,
            for each $y \in V - V'$, in order,
                if (($x$, $y$) is an edge)
                    add edge ($x$, $y$) to $E'$ and $y$ to $V'$
        if (no edges were added)
            return $T$
        $S$ = children of $S$ ordered consistently with the original vertex ordering
    }
}
```

Spanning Trees

- **Algorithm 12.3**: Depth-First Search
- Because of the line in Algorithm 12.3 where we retreat along an edge toward the initially chosen root, depth-first search is also called **backtracking.**



Depth-First Search for a Spanning Tree

This algorithm finds a spanning tree using the depth-first search method.

Input: A connected graph G with vertices ordered

$$v_1, \quad v_2, \quad \ldots, \quad v_n$$

Output: A spanning tree T

```
dfs(V, E) {
  // V' = vertices of spanning tree T; E' = edges of spanning tree T
  // v_1 is the root of the spanning tree
  $V' = \{v_1\}$
  $E' = \varnothing$
  $w = v_1$
  while (true) {
    while (there is an edge $(w, v)$ that when added to $T$ does not create a cycle
      in $T$) {
      choose the edge $(w, v_k)$ with minimum $k$ that when added to $T$
        does not create a cycle in $T$
      add $(w, v_k)$ to $E'$
      add $v_k$ to $V'$
      $w = v_k$
    }
    if ($w == v_1$)
      return $T$
    $w$ = parent of $w$ in $T$ // backtrack
  }
}
```

- Find a spanning tree for the graph *G* of Figure below.

We will use a method called **breadth-first search** (Algorithm 12.2). The idea of breadth-first search is to process all the vertices on a given level before moving to the next-higher level.

First, select an ordering, say *abcdefgh*, of the vertices of *G*. Select the first vertex *a* and label it the root. Let *T* consist of the single vertex *a* and no edges. Add to *T* all edges (a, x) and vertices on which they are incident, for $x = b$ to *h*, that do not produce a cycle when added to *T*. We would add to *T* edges (a, b), (a, c), and (a, g). (We could use either of the parallel edges incident on *a* and *g*.) Repeat this procedure with the vertices on level 1 by examining each in order:

b: Include (b, d).

c: Include (c, e).

g: None

Repeat this procedure with the vertices on level 2:

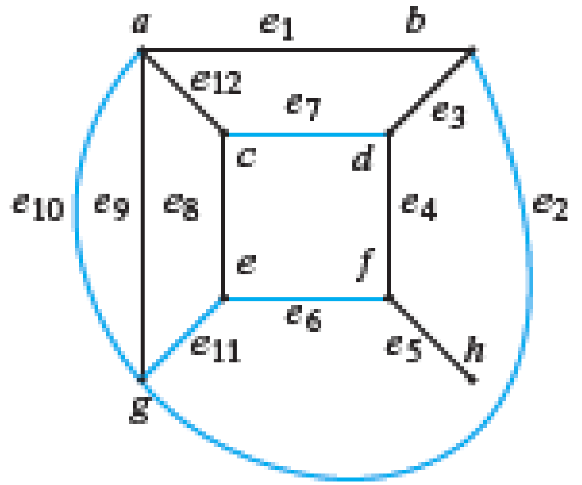
d: Include (d, f).

e: None

Repeat this procedure with the vertices on level 3:

f : Include (f, h).

Since no edges can be added to the single vertex *h* on level 4, the procedure ends. We have found the spanning tree shown in black color of the figure.

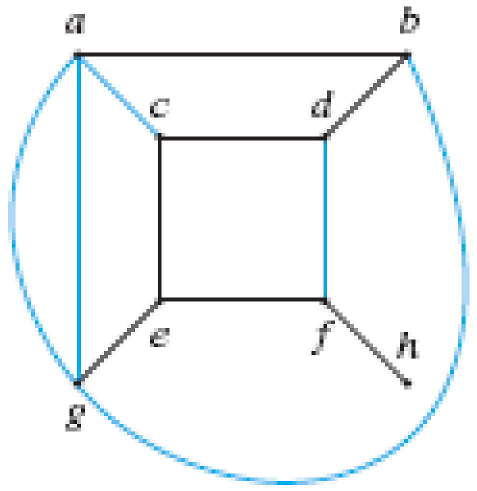


Spanning Trees

- ❑Use depth-first search (Algorithm 12.3) to find a spanning tree for the graph of Figure below with the vertex ordering *abcdefgh*.

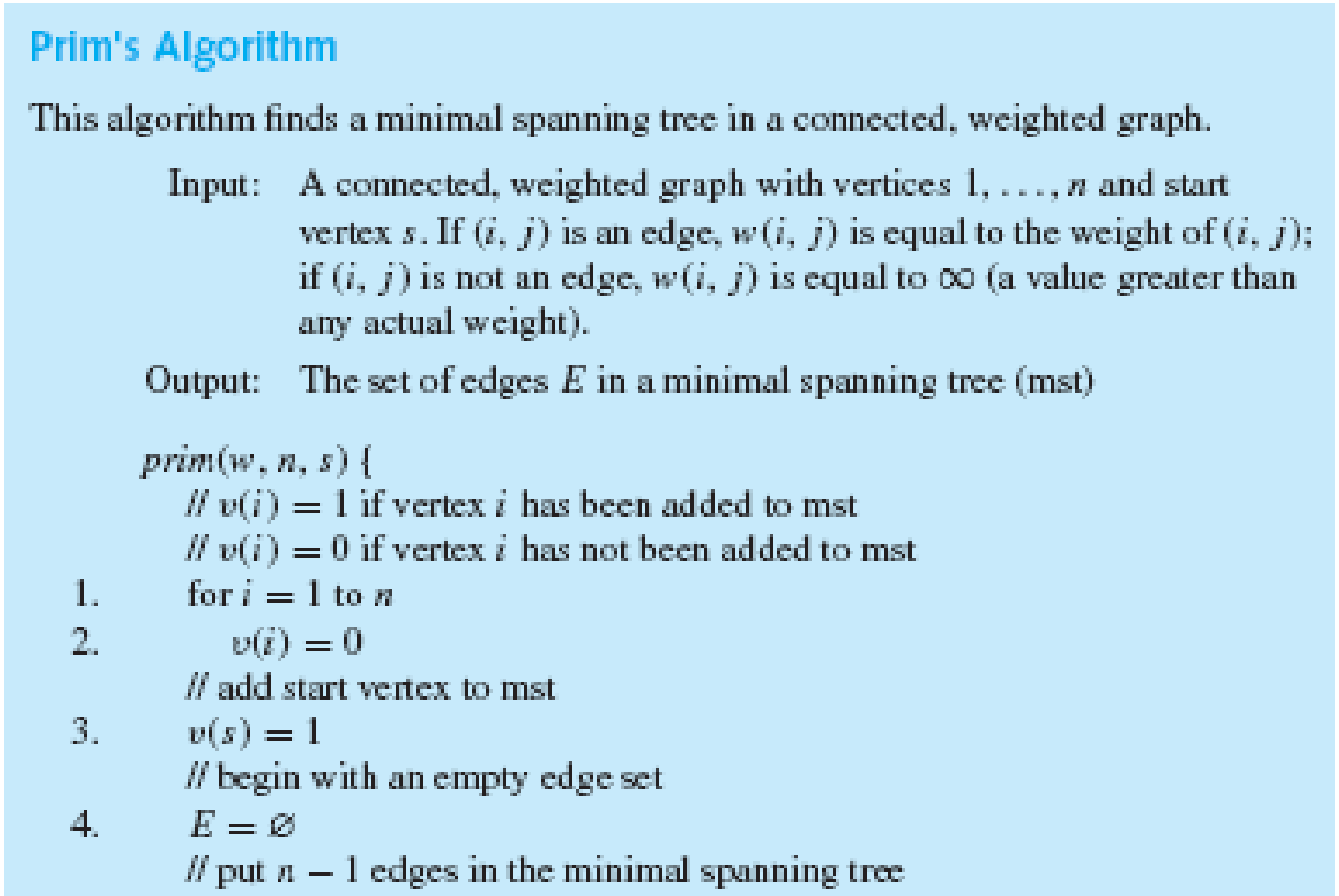
  We select the first vertex *a* and call it the root. Next, we add the edge (*a*, *x*), with minimal *x*, to our tree. In our case we add the edge (*a*, *b*).

  We repeat this process. We add the edges (*b*, *d*), (*d*, *c*), (*c*, e), (e, *f*), and (*f*, *h*). At this point, we cannot add an edge of the form (*h*, *x*), so we backtrack to the parent *f* of *h* and try to add an edge of the form (*f*, *x*). Again, we cannot add an edge of the form (*f*, *x*), so we backtrack to the parent e of *f* . This time we succeed in adding the edge (e, *g*). At this point, no more edges can be added, so we finally backtrack to the root and the procedure ends.



Minimal Spanning Trees

- **Definition 12.4**: Let G be a weighted graph. A **minimal spanning tree** of G is a spanning tree of G with minimum weight.



Prim's Algorithm

This algorithm finds a minimal spanning tree in a connected, weighted graph.

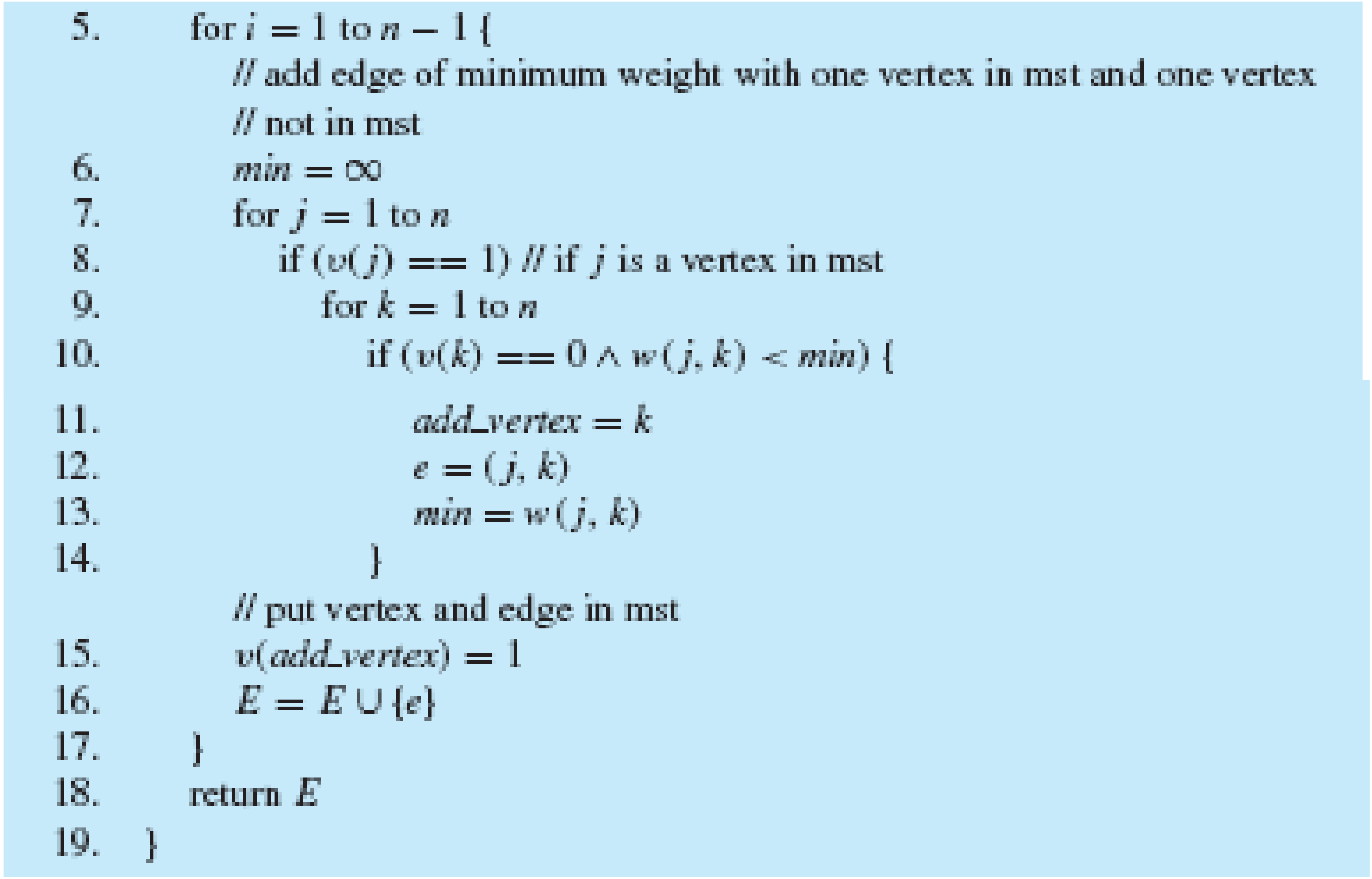
Input: A connected, weighted graph with vertices $1, \ldots, n$ and start vertex s. If (i, j) is an edge, $w(i, j)$ is equal to the weight of (i, j); if (i, j) is not an edge, $w(i, j)$ is equal to ∞ (a value greater than any actual weight).

Output: The set of edges E in a minimal spanning tree (mst)

```
      prim(w, n, s) {
         // v(i) = 1 if vertex i has been added to mst
         // v(i) = 0 if vertex i has not been added to mst
1.       for i = 1 to n
2.           v(i) = 0
         // add start vertex to mst
3.       v(s) = 1
         // begin with an empty edge set
4.       E = ∅
         // put n − 1 edges in the minimal spanning tree
```

Minimal Spanning Trees

- **Prim's Algorithm** furnishes an example of a **greedy algorithm.**
- A greedy algorithm is an algorithm that optimizes the choice at each iteration.
- The principle can be summarized as "doing the best locally."
- In Prim's Algorithm, since we want a minimal spanning tree, at each iteration we simply add an available edge with minimum weight.

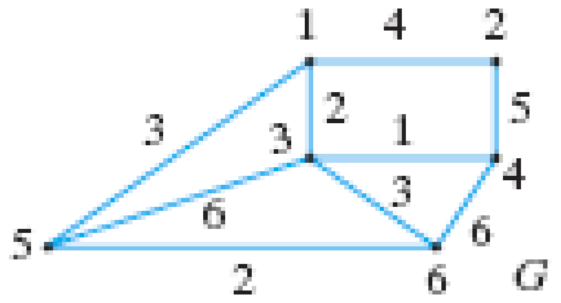


```
5.      for $i = 1$ to $n - 1$ {
          // add edge of minimum weight with one vertex in mst and one vertex
          // not in mst
6.        $min = \infty$
7.        for $j = 1$ to $n$
8.          if ($v(j) == 1$) // if $j$ is a vertex in mst
9.            for $k = 1$ to $n$
10.             if ($v(k) == 0 \wedge w(j, k) < min$) {
11.               $add\_vertex = k$
12.               $e = (j, k)$
13.               $min = w(j, k)$
14.             }
          // put vertex and edge in mst
15.       $v(add\_vertex) = 1$
16.       $E = E \cup \{e\}$
17.     }
18.     return $E$
19. }
```

- Show how Prim's Algorithm finds a minimal spanning tree for the graph of Figure below. Assume that the start vertex *s* is 1.

At line 3 we add vertex 1 to the minimal spanning tree. The first time we execute the for loop in lines 7–14, the edges with one vertex in the tree and one vertex not in the tree are

Edge	*Weight*
(1, 2)	4
(1, 3)	2
(1, 5)	3



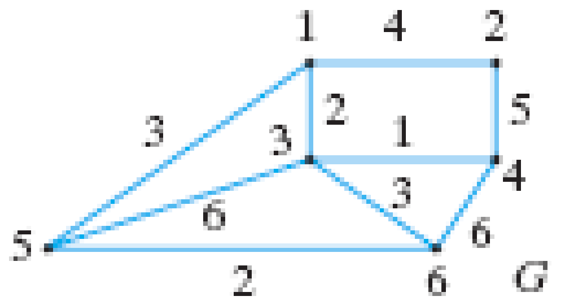
The edge (1, 3) with minimum weight is selected. At lines 15 and 16, vertex 3 is added to the minimal spanning tree and edge (1, 3) is added to *E*. The next time we execute the for loop in lines 7–14, the edges with one vertex in the tree and one vertex not in the tree are

Edge	*Weight*
(1, 2)	4
(1, 5)	3
(3, 4)	1
(3, 5)	6
(3, 6)	3

The edge (3, 4) with minimum weight is selected. At lines 15 and 16, vertex 4 is added to the minimal spanning tree and edge (3, 4) is added to *E*.

The next time we execute the for loop in lines 7–14, the edges with one vertex in the tree and one vertex not in the tree are

Edge	*Weight*
(1, 2)	4
(1, 5)	3
(2, 4)	5
(3, 5)	6
(3, 6)	3
(4, 6)	6



This time two edges have minimum weight 3. A minimal spanning tree will be constructed when either edge is selected. In this version, edge (1, 5) is selected. At lines 15 and 16, vertex 5 is added to the minimal spanning tree and edge (1, 5) is added to *E*.

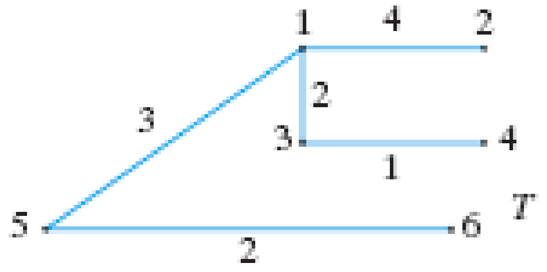
The next time we execute the for loop in lines 7–14, the edges with one vertex in the tree and one vertex not in the tree are

Edge	*Weight*
(1, 2)	4
(2, 4)	5
(3, 6)	3
(4, 6)	6
(5, 6)	2

The edge (5, 6) with minimum weight is selected. At lines 15 and 16, vertex 6 is added to the minimal spanning tree and edge (5, 6) is added to *E*.

The last time we execute the for loop in lines 7–14, the edges with one vertex in the tree and one vertex not in the tree are

Edge	*Weight*
(1, 2)	4
(2, 4)	5



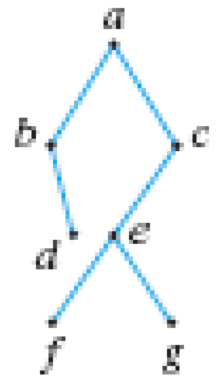
The edge (1, 2) with minimum weight is selected. At lines 15 and 16, vertex 2 is added to the minimal spanning tree and edge (1, 2) is added to *E*. The minimal spanning tree constructed is shown in Figure on the right.

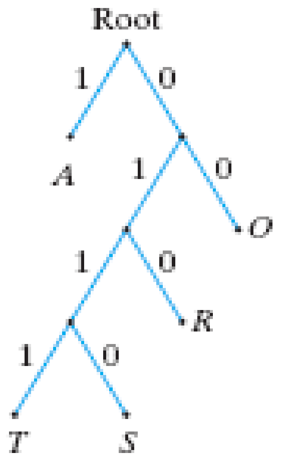
Binary Trees

- **Definition 12.5**: A **binary tree** is a rooted tree in which each vertex has either no children, one child, or two children. If a vertex has one child, that child is designated as either a left child or a right child (but not both). If a vertex has two children, one child is designated a left child and the other child is designated a right child.

❑In the binary tree of Figure on the right, vertex *b* is the left child of vertex *a* and vertex *c* is the right child of vertex *a*. Vertex *d* is the right child of vertex *b*; vertex *b* has no left child. Vertex *e* is the left child of vertex *c*; vertex *c* has no right child.

❑A tree that defines a Huffman code is a binary tree. For example, in the Huffman coding tree of Figure on the right, moving from a vertex to a left child corresponds to using the bit 1, and moving from a vertex to a right child corresponds to using the bit 0.



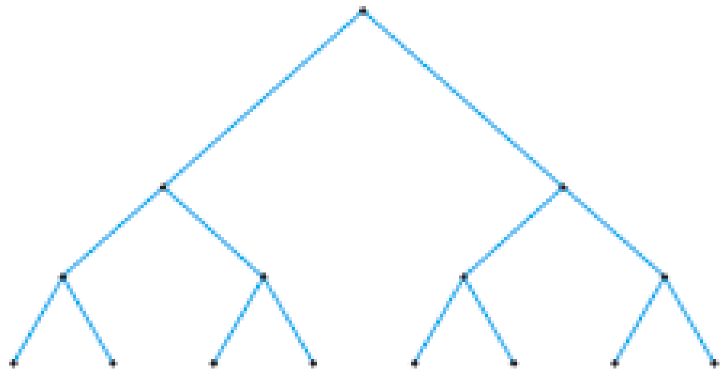


Binary Trees

- A **full binary tree** is a binary tree in which each vertex has either two children or zero children.
- **Theorem 12.3**: *If T is a full binary tree with i internal vertices, then T has i + 1 terminal vertices and 2i + 1 total vertices.*
- **Theorem 12.4**: *If a binary tree of height h has t terminal vertices, then*

$$\log_2 t \leq h. \quad (12.1)$$

❑ The binary tree in Figure below has height $h = 3$ and the number of terminals $t = 8$. For this tree, the inequality (12.1) becomes an equality.

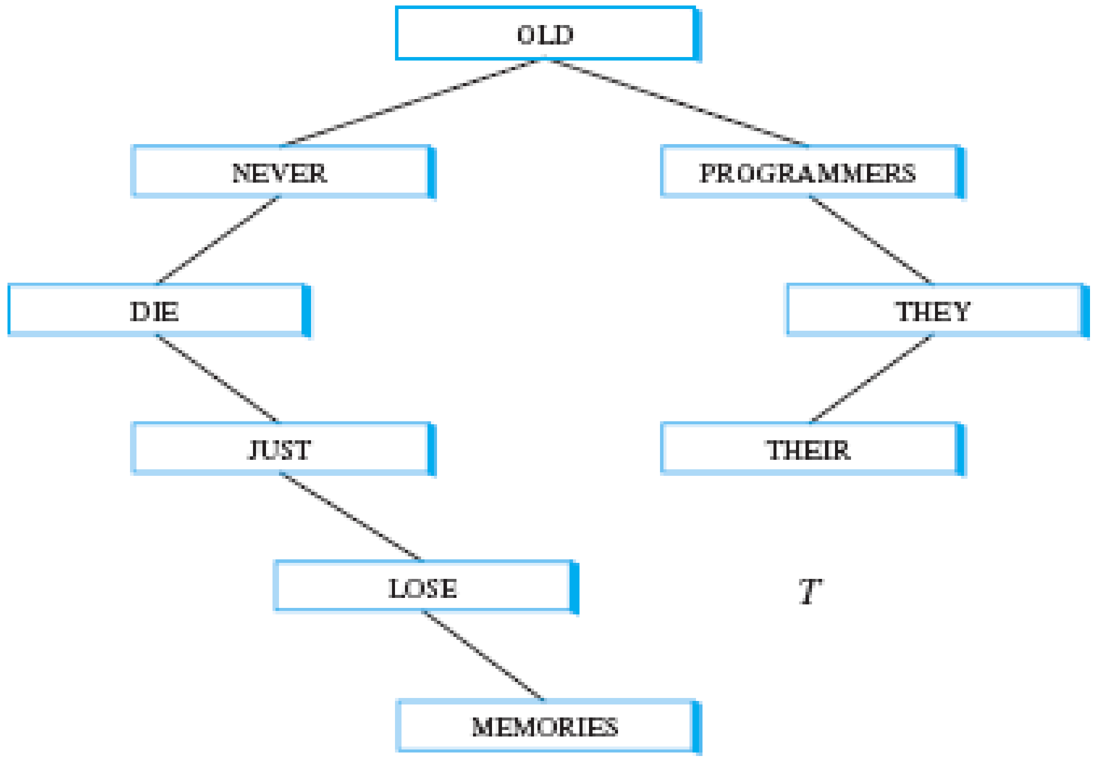


Binary Search Tree

- **Definition 12.6**: A **binary search tree** is a binary tree *T* in which data are associated with the vertices. The data are arranged so that, for *each* vertex *v* in *T*, each data item in the left subtree of *v* is less than the data item in *v*, and each data item in the right subtree of *v* is greater than the data item in *v*.

❑ The words

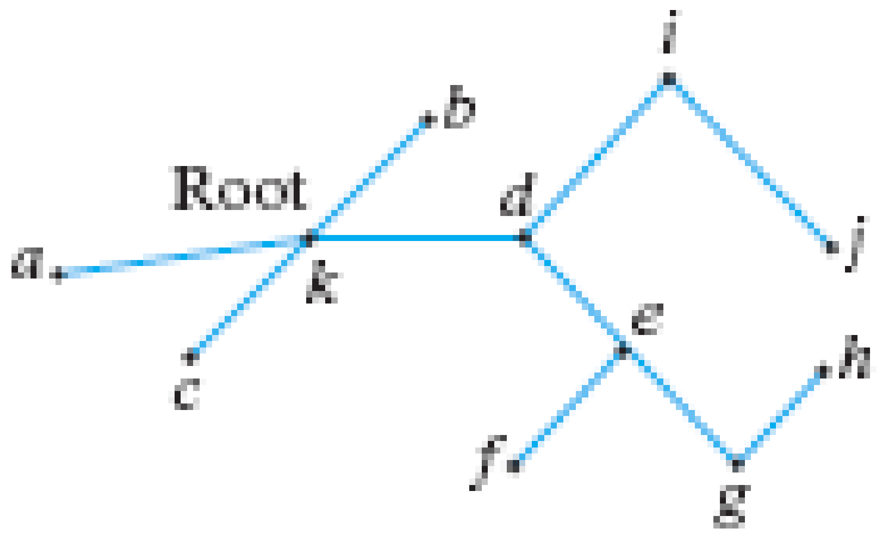
OLD PROGRAMMERS NEVER DIE
THEY JUST LOSE THEIR MEMORIES
may be placed in a binary search tree as shown in Figure on the right. Notice that for any vertex *v*, each data item in the left subtree of *v* is less than (i.e., precedes alphabetically) the data item in *v* and each data item in the right subtree of *v* is greater than the data item in *v*.



PRACTICE

PRACTICE 1

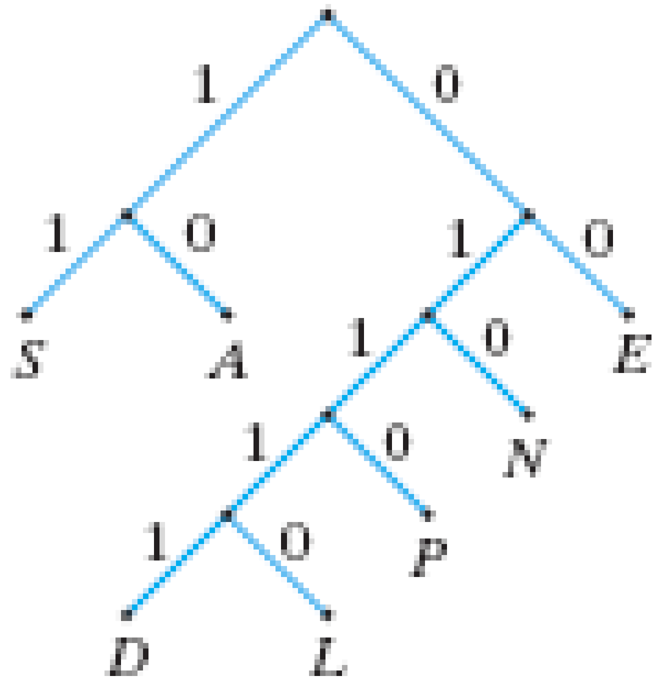
- Find the level of each vertex in the tree shown together with its height!



PRACTICE 2

- *Decode each bit string using the Huffman code given.*

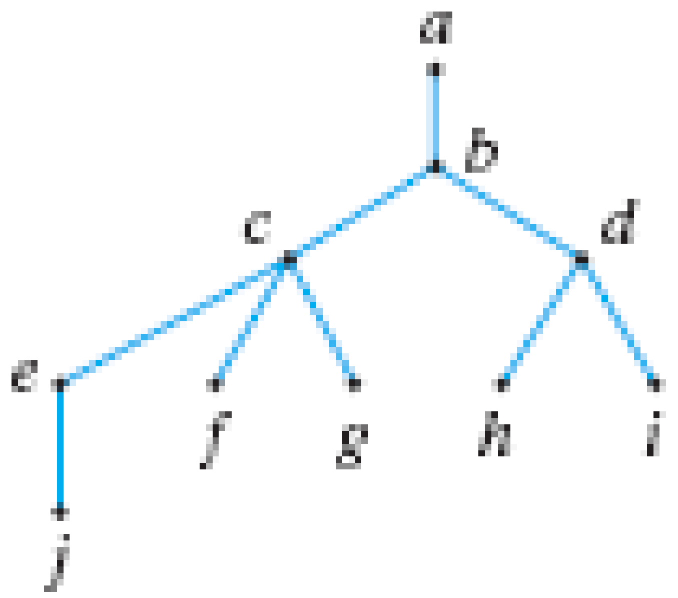
1. 011000010
2. 01110100110
3. 01111001001110



PRACTICE 3

- *Answer the questions below for the following tree.*

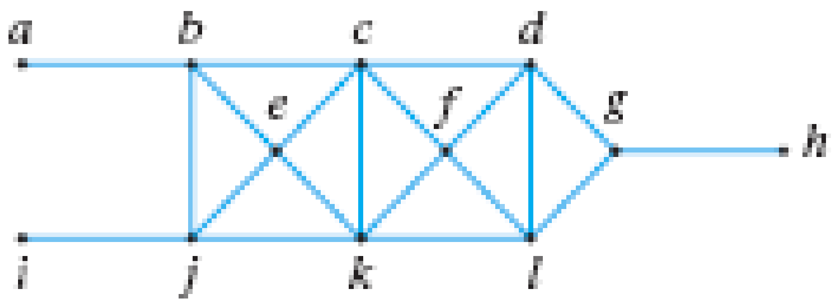
1. Find the parents of *c* and of *h*.
2. Find the ancestors of *c* and of *j* .
3. Find the children of *d* and of *e*.
4. Find the descendants of *c* and of e.
5. Find the siblings of *f* and of *h*.
6. Find the terminal vertices.
7. Draw the subtree rooted at *j*.



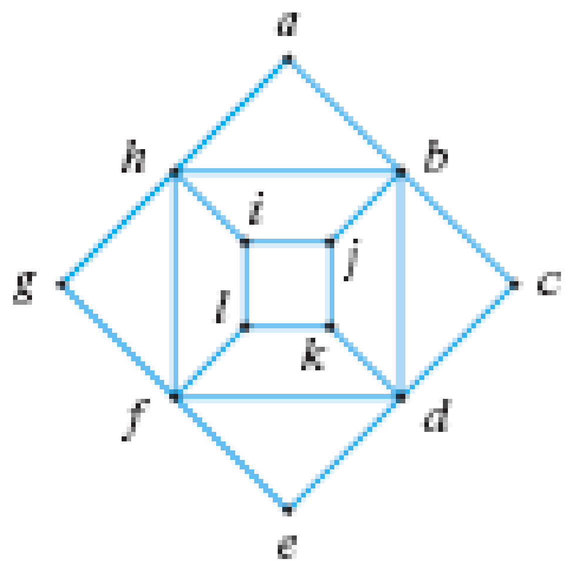
PRACTICE 4

- *In Exercises below, find a spanning tree for each graph (using breadth and depth-first search).*

1.



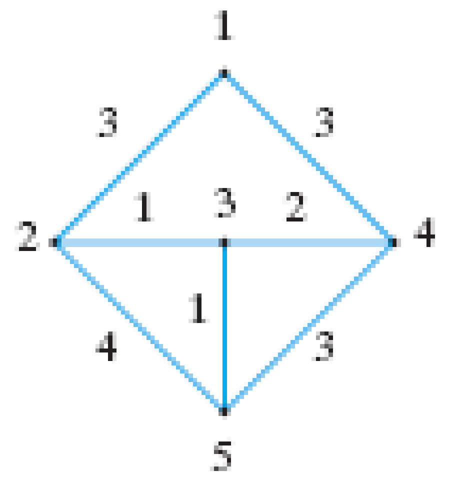
2.



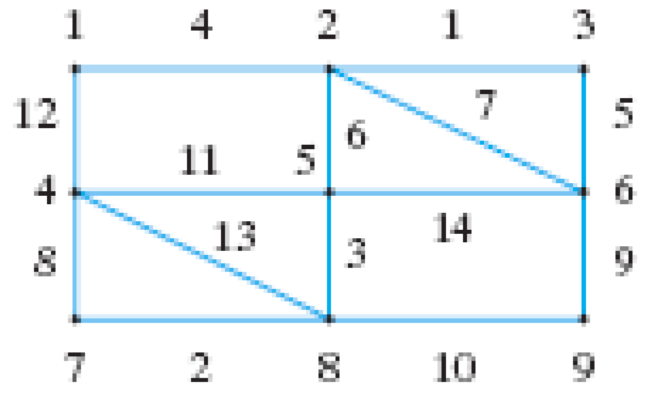
PRACTICE 5

- *In Exercises below, find the minimal spanning tree given by Algorithm 12.3 for each graph.*

1.



2.



NEXT WEEK'S OUTLINE

- Tree Traversals
- Decision Trees
- Trees Isomorphism
- Game Trees

REFERENCES

- Johnsonbaugh, R., 2005, *Discrete Mathematics*, New Jersey: Pearson Education, Inc.
- Rosen, Kenneth H., 2005, *Discrete Mathematics and Its Applications*, 6th edition, McGraw-Hill.
- Hansun, S., 2021, *Matematika Diskret Teknik*, Deepublish.
- Lipschutz, Seymour, Lipson, Marc Lars, *Schaum's Outline of Theory and Problems of Discrete Mathematics*, McGraw-Hill.
- Liu, C.L., 1995, *Dasar-Dasar Matematika Diskret*, Jakarta: Gramedia Pustaka Utama.
- Other offline and online resources.

Visi

Menjadi Program Studi Strata Satu Informatika **unggulan** yang menghasilkan lulusan **berwawasan internasional** yang **kompeten** di bidang Ilmu Komputer (*Computer Science*), **berjiwa wirausaha** dan **berbudi pekerti luhur**.

INFORMATIKA
UMN
UNIVERSITAS MULTIMEDIA NUSANTARA

Misi

1. Menyelenggarakan pembelajaran dengan teknologi dan kurikulum terbaik serta didukung tenaga pengajar profesional.
2. Melaksanakan kegiatan penelitian di bidang Informatika untuk memajukan ilmu dan teknologi Informatika.
3. Melaksanakan kegiatan pengabdian kepada masyarakat berbasis ilmu dan teknologi Informatika dalam rangka mengamalkan ilmu dan teknologi Informatika.