# Trees

Chapter 11



1

## Chapter Summary

- Graph Theory
- Introduction to Trees
- Applications of Trees
- Tree Traversal
- Spanning Trees
- Minimum Spanning Trees

2

# Introduction to Graphs

Sections 10.1-2-3-4

3

## Section Summary

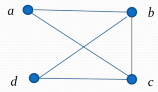
- Introduction Graphs: Terminology
- Graphs as Models
- Properties of Graphs and Fundamental Theorems

4

## Graphs

**Definition:** A *graph* $G = (V, E)$ consists of a nonempty set $V$ of *vertices* (or *nodes*) and a set $E$ of *edges*. Each edge has either one or two vertices associated with it, called its *endpoints*. An edge is said to *connect* its endpoints.

**Example:**
This is a graph with four vertices and five edges.



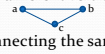

**Remarks**:

- The graphs we study here are unrelated to graphs of functions studied in Chapter 2.
- We have a lot of freedom when we draw a picture of a graph. All that matters is the connections made by the edges, not the particular geometry depicted. For example, the lengths of edges, whether edges cross, how vertices are depicted, and so on, do not matter
- A graph with an infinite vertex set is called an *infinite graph*. A graph with a finite vertex set is called a *finite graph*. We will restrict our attention to finite graphs.

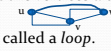
5

## Some Terminology

- In a *simple graph* each edge connects two different vertices and no two edges connect the same pair of vertices:



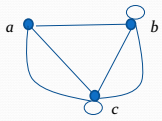

- *Multigraphs* may have multiple edges connecting the same two vertices. When $m$ different edges connect the vertices $u$ and $v$, we say that $\{u,v\}$ is an edge of *multiplicity* $m$.




- An edge that connects a vertex to itself is called a *loop*.


- A *pseudograph* may include loops, as well as multiple edges connecting the same pair of vertices.

**Example:**
This pseudograph has both multiple edges and loops.




**NOTE**: There is no standard terminology for graph theory. So, it is crucial that you understand the terminology being used whenever you read material about graphs.

6

## Directed Graphs

**Definition:** A *directed graph* (or *digraph*) $G = (V, E)$ consists of a nonempty set $V$ of *vertices* (or *nodes*) and a set $E$ of *directed edges* (or *arcs*). Each edge is associated with an ordered pair of vertices. The directed edge associated with the ordered pair $(u,v)$ is said to *start at* $u$ and *end at* $v$.

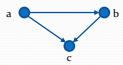
**Remark**:

- Graphs where the end points of an edge are not ordered are said to be *undirected graphs*.
- Conceptually, it may help to think of these terms as one way streets (directed edges) vs two way streets (undirected edges)

7

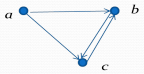
## Some Terminology

A *simple directed graph* has no loops and no multiple edges.




A *directed multigraph* may have multiple directed edges. When there are $m$ directed edges from the vertex $u$ to the vertex $v$, we say that $(u,v)$ is an edge of *multiplicity* $m$.

**Example**:
This is a directed multigraph with three vertices and four edges.




**Example**:
In this directed multigraph the multiplicity of $(a,b)$ is 1 and the multiplicity of $(b,c)$ is 2.


8

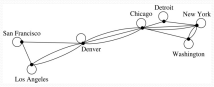
## Graph Models: Computer Networks

- When we build a graph model, we use the appropriate type of graph to capture the important features of the application.
- We illustrate this process using graph models of different types of computer networks. In all these graph models, the vertices represent data centers and the edges represent communication links.
- To model a computer network where we are only concerned whether two data centers are connected by a communications link, we use a simple graph. This is the appropriate type of graph when we only care whether two data centers are directly linked (and not how many links there may be) and all communications links work in both directions.

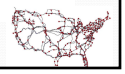

9

## Graph Models: Computer Networks (*continued*)

- To model a computer network where we care about the number of links between data centers, we use a multigraph.
- To model a computer network with diagnostic links at data centers, we use a pseudograph, as loops are needed.
- To model a network with multiple one-way links, we use a directed multigraph. Note that we could use a directed graph without multiple edges if we only care whether there is at least one link from a data center to another data center. Note: can you see something weird about this image? Hint: look carefully at New York





10

## Graph Terminology: Summary*

- To understand the structure of a graph and to build a graph model, we ask these questions:
  - Are the edges of the graph undirected or directed (or both)?
  - If the edges are undirected, are multiple edges present that connect the same pair of vertices? If the edges are directed, are multiple directed edges present?
  - Are loops present?

TABLE 1 Graph Terminology.

| *Type* | *Edges* | *Multiple Edges Allowed?* | *Loops Allowed?* |
|---|---|---|---|
| Simple graph | Undirected | No | No |
| Multigraph | Undirected | Yes | No |
| Pseudograph | Undirected | Yes | Yes |
| Simple directed graph | Directed | No | No |
| Directed multigraph | Directed | Yes | Yes |
| Mixed graph | Directed and undirected | Yes | Yes |

*Keep in mind that other authors may use somewhat different terminology

11

## Applications of Graphs

Graph theory can be used in models of:

- Social networks
- Communications networks
- Information networks
- Software design
- Transportation networks
- Biological networks

It's a challenge to find a subject to which graph theory has not yet been applied. Can you find an area without applications of graph theory? A few examples follow.

12

## Graph Models: Social Networks

Graphs can be used to model social structures based on different kinds of relationships between people or groups.

In a *social network*, vertices represent individuals or organizations and edges represent relationships between them.

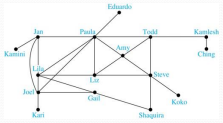
Useful graph models of social networks include:

- *friendship graphs* - undirected graphs where two people are connected if they are friends (in the real world, on Facebook, or in a particular virtual world, and so on.)
- *collaboration graphs* - undirected graphs where two people are connected if they collaborate in a specific way
- *influence graphs* - directed graphs where there is an edge from one person to another if the first person can influence the second person

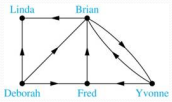
13

## Graph Models: Social Networks 2

**Example**: A friendship graph where two people are connected if they are Facebook friends. Elementary school teachers often use such graphs to study the dynamics of their classes.




**Example**: An influence graph; for example the graph says Brian and Yvonne influence each other, Fred influences Brian and is influenced by Deborah and Yvonne. Who influences the most other people? And what about Linda?




14

## Applications to Information Networks

Graphs can be used to model different types of networks that link different types of information.

In a *web graph*, web pages are represented by vertices and links are represented by directed edges.

- A web graph models the web at a particular time.
- We will explain how the web graph is used by search engines in Section 11.4.

In a *citation network*:

- Research papers in a particular discipline are represented by vertices.
- When a paper cites a second paper as a reference, there is an edge from the vertex representing this paper to the vertex representing the second paper.

15

## Software Design Applications

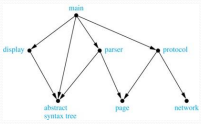
Graph models are extensively used in software design. We will introduce two such models here; one representing the dependency between the modules of a software application and the other representing restrictions in the execution of statements in computer programs.

When a top-down approach is used to design software, the system is divided into modules, each performing a specific task.

We use a *module dependency graph* to represent the dependency between these modules. These dependencies need to be understood before coding can be done.

- In a module dependency graph vertices represent software modules and there is an edge from one module to another if the second module depends on the first.

**Example**: The dependencies between the seven modules in the design of a web browser are represented by this module dependency graph.



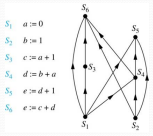

16

## Software Design Applications

We can use a directed graph called a *precedence graph* to represent which statements must have already been executed before we execute each statement.

- Vertices represent statements in a computer program
- There is a directed edge from a vertex to a second vertex if the second vertex cannot be executed before the first

**Example**: This precedence graph shows which statements must already have been executed before we can execute each of the six statements in the program.




17

## More Basic Terminology

**Definition 1**. Two vertices *u*, *v* in an undirected graph *G* are called ***adjacent*** (or ***neighbors***) in *G* if there is an edge *e* between *u* and *v*. Such an edge *e* is called *incident with* the vertices *u* and *v* and *e* is said to *connect u* and *v*.

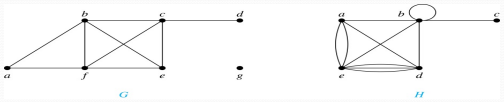
**Definition 2**. The set of all neighbors of a vertex *v* of $G = (V, E)$, denoted by $N(v)$, is called the ***neighborhood*** of *v*. If *A* is a subset of *V*, we denote by $N(A)$ the set of all vertices in *G* that are adjacent to at least one vertex in *A*. So, $N(A) = \bigcup_{v \in A} N(v)$.

**Definition 3**. The ***degree of a vertex in an undirected graph*** is the number of edges incident with it, except that a loop at a vertex contributes two to the degree of that vertex. The degree of the vertex *v* is denoted by $\deg(v)$.

18

## Degrees and Neighborhoods of Vertices

**Example**: What are the degrees and neighborhoods of the vertices in the graphs $G$ and $H$?




**Solution**:

$G$: $\deg(a) = 2$, $\deg(b) = \deg(c) = \deg(f) = 4$, $\deg(d) = 1$, $\deg(e) = 3$, $\deg(g) = 0$.
$N(a) = \{b, f\}$, $N(b) = \{a, c, e, f\}$, $N(c) = \{b, d, e, f\}$, $N(d) = \{c\}$, $N(e) = \{b, c, f\}$, $N(f) = \{a, b, c, e\}$, $N(g) = \varnothing$.

$H$: $\deg(a) = 4$, $\deg(b) = \deg(e) = 6$, $\deg(c) = 1$, $\deg(d) = 5$.
$N(a) = \{b, d, e\}$, $N(b) = \{a, b, c, d, e\}$, $N(c) = \{b\}$, $N(d) = \{a, b, e\}$, $N(e) = \{a, b, d\}$.

19

## Degrees of Vertices

**Theorem (*Handshaking Theorem*)**: If $G = (V,E)$ is an undirected graph with $m$ edges, then

$$2m = \sum_{v \in V} \deg(v)$$

***Proof***:

Each edge contributes twice to the degree count of all vertices. Hence, both the left-hand and right-hand sides of this equation equal twice the number of edges.

◄

*Think about the graph where vertices represent the people at a party and an edge connects two people who have shaken hands. Also note that the sum is EVEN (2m).*

20

## Handshaking Theorem Applications

We now give two examples illustrating the usefulness of the handshaking theorem.

**Example**: How many edges are there in a graph with 10 vertices of degree six?

**Solution**: Because the sum of the degrees of the vertices is $6 \cdot 10 = 60$, the handshaking theorem tells us that $2m = 60$. So the number of edges $m = 30$.

**Example**: If a graph has 5 vertices, can each vertex have degree 3?

**Solution**: This is not possible by the handshaking theorem, because the sum of the degrees of the vertices $3 \cdot 5 = 15$ is odd.

21

## Degree of Vertices (*continued*)

**Theorem 2:** An undirected graph has an even number of vertices of odd degree.

***Proof***: Let $V_1$ be the vertices of even degree and $V_2$ be the vertices of odd degree in an undirected graph $G = (V, E)$ with $m$ edges. Then

even → $2m = \sum_{v \in V} \deg(v) = \sum_{v \in V_1} \deg(v) + \sum_{v \in V_2} \deg(v).$

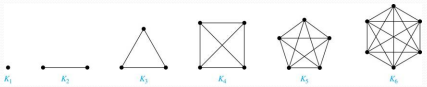
must be even since $\deg(v)$ is even for each $v \in V_1$

This sum must be even because $2m$ is even and the sum of the degrees of the vertices of even degrees is also even. Because this is the sum of the degrees of all vertices of odd degree in the graph, there must be an even number of such vertices.

22

## Special Types of Simple Graphs: Complete Graphs

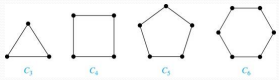
A *complete graph on n vertices*, denoted by $K_n$, is the simple graph that contains exactly one edge between each pair of distinct vertices.



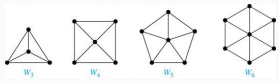
23

## Special Types of Simple Graphs: Cycles and Wheels

A *cycle* $C_n$ for $n \geq 3$ consists of $n$ vertices $v_1, v_2, \cdots, v_n$, and edges $\{v_1, v_2\}, \{v_2, v_3\}, \cdots, \{v_{n-1}, v_n\}, \{v_n, v_1\}$.



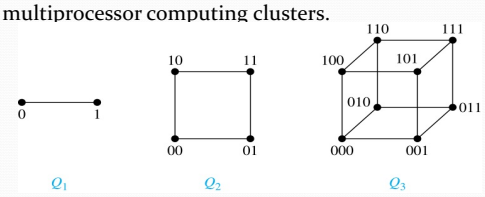
A *wheel* $W_n$ is obtained by adding an additional vertex to a cycle $C_n$ for $n \geq 3$ and connecting this new vertex to each of the $n$ vertices in $C_n$ by new edges.



24

## Special Types of Simple Graphs: *n*-Cubes

An *n-dimensional hypercube*, or *n-cube*, $\boldsymbol{Q_n}$, is a graph with $2^n$ vertices representing all bit strings of length $n$, where there is an edge between two vertices that differ in exactly one bit position. Hypercubes are commonly used as the interconnection geometry of multiprocessor computing clusters.



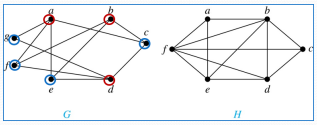

25

## Bipartite Graphs

**Definition:** A simple graph $G$ is *bipartite* if $V$ can be partitioned into two *disjoint* subsets $V_1$ and $V_2$ such that every edge connects a vertex in $V_1$ and a vertex in $V_2$. In other words, there are no edges which connect two vertices in $V_1$ or in $V_2$.

An equivalent definition of a bipartite graph is: a graph is bipartite when it is possible to color the vertices red or blue (or any two distinct colors) so that no two adjacent vertices are the same color.

$G$ is bipartite



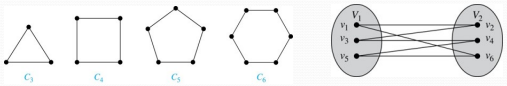

$H$ is not bipartite since if we color $a$ red, then the adjacent vertices $f$ and $b$ must both be blue.

26

## Bipartite Graphs

**Example**: Show that $C_6$ is bipartite.

**Solution**: We can partition the vertex set into $V_1 = \{v_1, v_3, v_5\}$ and $V_2 = \{v_2, v_4, v_6\}$ so that every edge of $C_6$ connects a vertex in $V_1$ and $V_2$ .



**Example**: Show that $C_3$ is not bipartite.

**Solution**: If we divide the vertex set of $C_3$ into two nonempty sets, one of the two must contain two vertices. But in $C_3$ every vertex is connected to every other vertex. Therefore, the two vertices in the same partition are connected. Hence, $C_3$ is not bipartite. What can you say about $C_4$ and $C_5$?

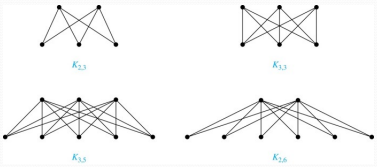
***Solution:*** *$C_4$* ***is*** Bipartite and *$C_5$* ***is not*** Bipartite. Any conjectures about which *$C_n$ are bipartite and which are not?*

27

## Complete Bipartite Graphs

**Definition:** A ***complete bipartite graph*** $K_{m,n}$ is a graph that has its vertex set partitioned into two subsets $V_1$ of size $m$ and $V_2$ of size $n$ such that there is an edge from every vertex in $V_1$ to every vertex in $V_2$.

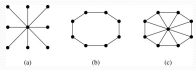
**Example**: Here are four complete bipartite graphs:



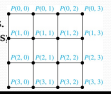

28

## Special Types of Graphs and Computer Network Architecture

Various special graphs play an important role in the design of computer networks.




- Some local area networks use a *star topology*, which is a complete bipartite graph $K_{1,n}$, as shown in (a). All devices are connected to a central control device.
- Other local networks are based on a *ring topology*, where each device is connected to exactly two others using $C_n$, as illustrated in (b). Messages may be sent around the ring.
- Others, as illustrated in (c), use a $W_n$ – based topology, combining the features of a star topology and a ring topology.
- Various special graphs also play a role in parallel processing where processors need to be interconnected as one processor may need the output generated by another.
  - The *n-dimensional hypercube*, or *n-cube*, $Q_n$, is a common way to connect processors in parallel, e.g., Intel Hypercube.
  - Another common method is the *mesh* network, illustrated here for 16 devices.
  - Mesh networks are also used by several brands of wireless speakers, e.g., Sonos.
  - Some WIFI Network extenders also use the mesh design

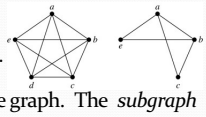


29

## New Graphs from Old

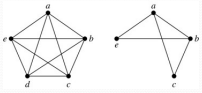
**Definition:** A *subgraph of a graph* $G = (V,E)$ is a graph $(W,F)$, where $W \subset V$ and $F \subset E$. A subgraph $H$ of $G$ is a proper subgraph of $G$ if $H \neq G$.

**Example**: $K_5$ and one of its subgraphs.




**Definition:** Let $G = (V, E)$ be a simple graph. The *subgraph induced* by a subset $W$ of the vertex set $V$ is the graph $(W,F)$, where the edge set $F$ contains an edge in $E$ if and only if both endpoints are in $W$.

**Example**: Here we show $K_5$ and the subgraph induced by $W = \{a,b,c,e\}$.

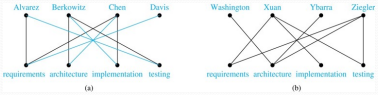



30

## Bipartite Graphs and Matchings

Bipartite graphs are used to model applications that involve matching the elements of one set to elements in another, for example:

*Job assignments* - vertices represent the jobs and the employees, edges link employees with those jobs they have been trained to do. A common goal is to match jobs to employees so that the most jobs are done.




*Marriage* - vertices represent the men and the women and edges link a man and a woman if they are an acceptable spouse. We may wish to find the largest number of possible marriages.

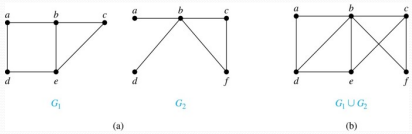
*See the text for more about matchings in bipartite graphs.*

31

## New Graphs from Old

**Definition**: The *union* of two simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the simple graph with vertex set $V_1 \cup V_2$ and edge set $E_1 \cup E_2$. The union of $G_1$ and $G_2$ is denoted by $G_1 \cup G_2$.

**Example**:




32

## Representing Graphs: Adjacency Lists

**Definition**: An *adjacency list* can be used to represent a graph with no multiple edges by specifying the vertices that are adjacent to each vertex of the graph.

**Example**:

TABLE 1 An Adjacency List for a Simple Graph.

| Vertex | Adjacent Vertices |
|---|---|
| $a$ | $b, c, e$ |
| $b$ | $a$ |
| $c$ | $a, d, e$ |
| $d$ | $c, e$ |
| $e$ | $a, c, d$ |

**Example**:

TABLE 2 An Adjacency List for a Directed Graph.

| Initial Vertex | Terminal Vertices |
|---|---|
| $a$ | $b, c, d, e$ |
| $b$ | $b, d$ |
| $c$ | $a, c, e$ |
| $d$ | |
| $e$ | $b, c, d$ |

33

## Representation of Graphs: Adjacency Matrices

**Definition**: Suppose that $G = (V, E)$ is a simple graph where $|V| = n$. Arbitrarily list the vertices of $G$ as $v_1, v_2, \ldots, v_n$. The *adjacency matrix* $\mathbf{A}_G$ of $G$, with respect to the listing of vertices, is the $n \times n$ zero-one matrix with 1 as its $(i, j)$th entry when $v_i$ and $v_j$ are adjacent, and 0 as its $(i, j)$th entry when they are not adjacent.

- In other words, if the graphs adjacency matrix is $\mathbf{A}_G = [a_{ij}]$, then

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$

34

## Adjacency Matrices

**Example**:


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

*The ordering of vertices is a, b, c, d.*


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

*The ordering of vertices is a, b, c, d.*

When a graph is sparse, that is, it has few edges relatively to the total number of possible edges, it is much more efficient to represent the graph using an adjacency list than an adjacency matrix. But for a dense graph, which includes a high percentage of possible edges, an adjacency matrix is preferable.

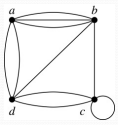
**Note**: Do you notice anything special about these matrices?
**SOLUTION:** The adjacency matrix of a simple graph is symmetric, i.e., $a_{ij} = a_{ji}$
Also, since there are no loops, each diagonal entry $a_{ij}$ for $i = 1, 2, 3, \ldots, n$, is 0.

35

## Adjacency Matrices (*continued*)

- Adjacency matrices can also be used to represent graphs with loops and multiple edges.
- A loop at the vertex $v_i$ is represented by a 1 at the $(i, i)^{\text{th}}$ position of the matrix.
- **When multiple edges connect the same pair of vertices** $v_i$ and $v_j$, (or if multiple loops are present at the same vertex), **the $(i, j)$th entry equals the number of edges connecting the pair of vertices**.

**Example**: Here is the adjacency matrix of the pseudograph shown using the ordering of vertices $a, b, c, d$.




$$\begin{bmatrix} 0 & 3 & 0 & 2 \\ 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 2 & 1 & 2 & 0 \end{bmatrix}$$

36

## Adjacency Matrices (*continued*)

- Adjacency matrices can also be used to represent directed graphs. The matrix for a directed graph $G = (V, E)$ has a 1 in its $(i, j)$th position if there is an edge from $v_i$ to $v_j$, where $v_1, v_2, \ldots v_n$ is a list of the vertices.
  - In other words, if the graphs adjacency matrix is $\mathbf{A}_G = [a_{ij}]$, then
$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$
  - The adjacency matrix for a directed graph does not have to be symmetric, because there may not be an edge from $v_i$ to $v_j$, when there is an edge from $v_j$ to $v_i$.
  - To represent directed multigraphs, the value of $a_{ij}$ is the number of edges connecting $v_i$ to $v_j$.

37

## Representation of Graphs: Incidence Matrices

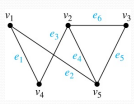
**Definition**: Let $G = (V, E)$ be an undirected graph with vertices $v_1, v_2, \ldots v_n$ and edges $e_1, e_2, \ldots e_m$. The incidence matrix with respect to the ordering of $V$ and $E$ is the $n \times m$ matrix $\mathbf{M} = [m_{ij}]$, where

$$m_{ij} = \begin{cases} 1 & \text{when edge } e_j \text{ is incident with } v_i, \\ 0 & \text{otherwise.} \end{cases}$$

38

## Incidence Matrices (*continued*)

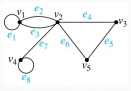
**Example**: Simple Graph and Incidence Matrix




| | $e_1$ | $e_2$ | $e_3$ | $e_4$ | $e_5$ | $e_6$ |
|---|---|---|---|---|---|---|
| $v_1$ | 1 | 1 | 0 | 0 | 0 | 0 |
| $v_2$ | 0 | 0 | 1 | 1 | 0 | 1 |
| $v_3$ | 0 | 0 | 0 | 0 | 1 | 1 |
| $v_4$ | 1 | 0 | 1 | 0 | 0 | 0 |
| $v_5$ | 0 | 1 | 0 | 1 | 1 | 0 |

*The rows going from top to bottom represent $v_1$ through $v_5$ and the columns going from left to right represent $e_1$ through $e_6$.*

**Example**: Pseudograph and Incidence Matrix




| | $e_1$ | $e_2$ | $e_3$ | $e_4$ | $e_5$ | $e_6$ | $e_7$ | $e_8$ |
|---|---|---|---|---|---|---|---|---|
| $v_1$ | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| $v_2$ | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| $v_3$ | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| $v_4$ | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| $v_5$ | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

*The rows going from top to bottom represent $v_1$ through $v_5$ and the columns going from left to right represent $e_1$ through $e_8$.*

39

## Paths

**Informal Definition:** A *path* is a sequence of edges that begins at a vertex of a graph and travels from vertex to vertex along edges of the graph. As the path travels along its edges, it visits the vertices along this path, that is, the endpoints of these edges.

**Applications**: Numerous problems can be modeled with paths formed by traveling along edges of graphs such as:

- determining whether a message can be sent between two computers.
- efficiently planning routes for mail delivery.

40

## Paths

**Definition:** Let $n$ be a nonnegative integer and $G$ an undirected graph. A *path* of *length* $n$ from $u$ to $v$ in $G$ is a sequence of $n$ edges $e_1, \ldots, e_n$ of $G$ for which there exists a sequence $x_0 = u, x_1, \ldots, x_{n-1}, x_n = v$ of vertices such that $e_i$ has, for $i = 1, \ldots, n$, the endpoints $x_{i-1}$ and $x_i$.

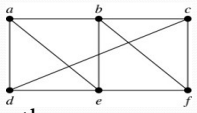
- When the graph is simple, we denote this path by its vertex sequence $x_0, x_1, \ldots, x_n$ (since listing the vertices uniquely determines the path).
- The path is a *circuit* if it begins and ends at the same vertex ($u = v$) and has length greater than zero.
- The path or circuit is said to *pass through* the vertices $x_1, x_2, \ldots, x_{n-1}$ and *traverse* the edges $e_1, \ldots, e_n$.
- A path or circuit is *simple* if it does not contain the same edge more than once.

This terminology is readily extended to directed graphs.

41

## Paths

**Example**: In this simple graph:




- $a, d, c, f, e$ is a simple path of length 4.
- $d, e, c, a$ is not a path because $e$ is not connected to $c$.
- $b, c, f, e, b$ is a circuit of length 4.
- $a, b, e, d, a, b$ is a path of length 5, but it is not a simple path because it traverses edge $(a, b)$ twice.

42

# Introduction to Trees

Section 11.1

43

## Section Summary

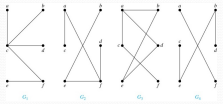
- Introduction to Trees
- Rooted Trees
- Trees as Models
- Properties of Trees

44

## Trees

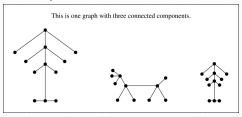
**Definition**: A *tree* is a connected undirected graph with no simple circuits.

**Example**: Which of these graphs are trees?



**Solution**: $G_1$ and $G_2$ are trees - both are connected and have no simple circuits. Because $e, b, a, d, e$ is a simple circuit, $G_3$ is not a tree. $G_4$ is not a tree because it is not connected.

**Definition**: A *forest* is a graph that has no simple circuit, but is not connected. Each of the connected components in a forest is a tree.




45

## Trees

- Consider a tree T. Clearly, there can only one simple path between any two vertices of T ; otherwise, if there were more than one path, the two paths would form a cycle. Also:
  a) Suppose there is no edge {u,v} in T and we add the edge e={u,v} to T. Then the simple path from u to v in T and e will form a cycle; hence T is no longer a tree.
  b) On the other hand, suppose there is an edge e = {u,v} in T, and we delete e from T. Then T is no longer connected (since there cannot be a path from u to v); hence T is no longer a tree.

46

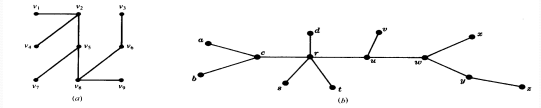
## Trees

- Theorem: Assume *n* is finite and:

Let G be a graph with n > 1 vertices. Then the following are equivalent:

i. G is a tree.
ii. G is cycle-free and has n – 1 edges.
iii. G is connected and has n–1 edges.

This theorem also tells us that a finite tree T with *n* vertices *must* have *n* – 1 edges. For example, the tree (a) has 9 vertices and 8 edges, and the tree (b) has 13 vertices and 12 edges.



47

## Trees (*continued*)

**Theorem**: An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

***Proof***: Assume that *T* is a tree. Then *T* is connected with no simple circuits. Hence, if *x* and y are distinct vertices of *T*, there is a simple path between them (by Theorem 1 of Section 10.4). This path must be unique - for if there were a second path, there would be a simple circuit in *T* (by Exercise 59 of Section 10.4). Hence, there is a unique simple path between any two vertices of a tree.

Now assume that there is a unique simple path between any two vertices of a graph *T*. Then *T* is connected because there is a path between any two of its vertices. Furthermore, *T* can have no simple circuits since if there were a simple circuit, there would be two paths between some two vertices.

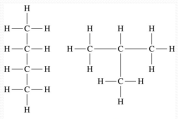
Hence, a graph with a unique simple path between any two vertices is a tree ◄

48

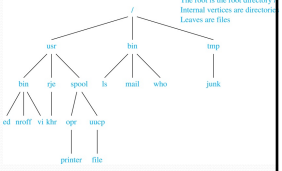
# Trees as Models

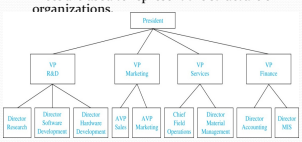
Arthur Cayley
(1821-1895)

- Trees are used as models in computer science, chemistry, geology, botany, psychology, and many other areas.
- Trees were introduced by the mathematician Cayley in 1857 in his work counting the number of isomers of saturated hydrocarbons. The two isomers of butane are shown at the right.
- The organization of a computer file system into directories, subdirectories, and files is naturally represented as a tree.
- Trees are used to represent the structure of organizations.



Butane Isobutane



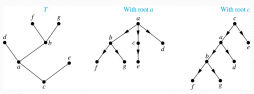




49

# Rooted Trees

**Definition**: A *rooted tree* is a tree in which one vertex has been designated as the *root* and every edge is directed away from the root.

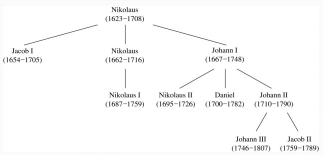
An unrooted tree is converted into different rooted trees when different vertices are chosen as the root.




50

# Rooted Tree Terminology

- Terminology for rooted trees is a mix from botany and genealogy (such as this family tree of the Bernoulli family of mathematicians).




- If $v$ is a vertex of a rooted tree other than the root, the *parent* of $v$ is the unique vertex $u$ such that there is a directed edge from $u$ to $v$. When $u$ is a parent of $v$, $v$ is called a *child* of $u$. Vertices with the same parent are called *siblings*.
- The *ancestors* of a vertex are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root. The *descendants* of a vertex $v$ are those vertices that have $v$ as an ancestor.
- A vertex of a rooted tree with no children is called a *leaf*. Vertices that have children are called *internal vertices*.
- If $a$ is a vertex in a tree, the *subtree* with $a$ as its root is the subgraph of the tree consisting of $a$ and its descendants and all edges incident to these descendants.

51

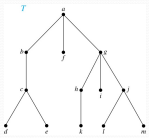
# Terminology for Rooted Trees

**Example**: In the rooted tree $T$ (with root $a$):

(i) Find the parent of $c$, the children of $g$, the siblings of $h$, the ancestors of $e$, and the descendants of $b$.

(ii) Find all internal vertices and all leaves.

(iii) What is the subtree rooted at $G$?

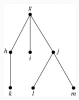


**Solution**:

(i) The parent of $c$ is $b$. The children of $g$ are $h$, $i$, and $j$. The siblings of $h$ are $i$ and $j$. The ancestors of $e$ are $c$, $b$, and $a$. The descendants of $b$ are $c$, $d$, and $e$.

(ii) The internal vertices are $a$, $b$, $c$, $g$, $h$, and $j$. The leaves are $d$, $e$, $f$, $i$, $k$, $l$, and $m$.

(iii) The subtree rooted at $g$ is to the right.

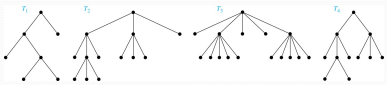


52

# *m*-ary Rooted Trees

**Definition**: A rooted tree is called an *m-ary tree* if every internal vertex has no more than $m$ children. The tree is called a *full m-ary tree* if every internal vertex has exactly $m$ children. An $m$-ary tree with $m = 2$ is called a *binary* tree.

**Example**: Are the following rooted trees full $m$-ary trees for some positive integer $m$?



**Solution**: $T_1$ is a full binary tree because each of its internal vertices has two children. $T_2$ is a full 3-ary tree because each of its internal vertices has three children. In $T_3$ each internal vertex has five children, so $T_3$ is a full 5-ary tree. $T_4$ is not a full $m$-ary tree for any m because some of its internal vertices have two children and others have three children.

53

# Ordered Rooted Trees

**Definition**: An *ordered rooted tree* is a rooted tree where the children of each internal vertex are ordered.

- We draw ordered rooted trees so that the children of each internal vertex are shown in order from left to right.

**Definition**: A *binary tree* is an ordered rooted tree where each internal vertex has at most two children. If an internal vertex of a binary tree has two children, the first is called the *left child* and the second the *right child*. The tree rooted at the left child of a vertex is called the *left subtree* of this vertex, and the tree rooted at the right child of a vertex is called the *right subtree* of this vertex.

**Example**: Consider the binary tree $T$.

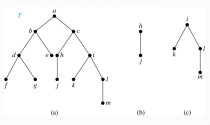
(i) What are the left and right children of $d$?

(ii) What are the left and right subtrees of $c$?

**Solution**:

(i) The left child of $d$ is $f$ and the right child is $g$.

(ii) The left and right subtrees of $c$ are displayed in (b) and (c).



54

## Properties of Trees

**Theorem 2**: A tree with $n$ vertices has $n - 1$ edges.

***Proof* (*by mathematical induction*):**
*BASIS STEP*: When $n = 1$, a tree with one vertex has no edges. Hence, the theorem holds when $n = 1$.
*INDUCTIVE STEP*: Assume that every tree with $k$ vertices has $k - 1$ edges.
Suppose that a tree $T$ has $k + 1$ vertices and that $v$ is a leaf of $T$. Let $w$ be the parent of $v$. Removing the vertex $v$ and the edge connecting $w$ to $v$ produces a tree $T'$ with $k$ vertices. By the inductive hypothesis, $T'$ has $k - 1$ edges. Because $T$ has one more edge than $T'$, we see that $T$ has $k$ edges. This completes the inductive step. ◄

55

## Counting Vertices in Full *m*-Ary Trees

**Theorem 3**: A full $m$-ary tree with $i$ internal vertices has $n = mi + 1$ vertices.

***Proof*** : Every vertex, except the root, is the child of an internal vertex. Because each of the $i$ internal vertices has $m$ children, there are $mi$ vertices in the tree other than the root. Hence, the tree contains $n = mi + 1$ vertices. ◄

56

## Counting Vertices in Full *m*-Ary Trees (*continued*)

**Theorem 4**: A full $m$-ary tree with

(*i*) $n$ vertices has $i = (n - 1)/m$ internal vertices and $l = [(m - 1)n + 1]/m$ leaves,

(*ii*) $i$ internal vertices has $n = mi + 1$ vertices and $l = (m - 1)i + 1$ leaves,

(*iii*) $l$ leaves has $n = (ml - 1)/(m - 1)$ vertices and $i = (l - 1)/(m - 1)$ internal vertices.

*proofs of parts (ii) and (iii) are left as exercises*

***Proof* (*of part i*):** Solving for $i$ in $n = mi + 1$ (from Theorem 3) gives $i = (n - 1)/m$. Since each vertex is either a leaf or an internal vertex, $n = l + i$. By solving for $l$ and using the formula for $i$, we see that

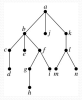
$$l = n - i = n - (n - 1)/m = [(m - 1)n + 1]/m.$$ ◄

57

## Level of vertices and height of trees

- When working with trees, we often want to have rooted trees where the subtrees at each vertex contain paths of approximately the same length.
- To make this idea precise we need some definitions:
  - The *level* of a vertex $v$ in a rooted tree is the length of the unique path from the root to this vertex.
  - The *height* of a rooted tree is the maximum of the levels of the vertices.

**Example**:
(*i*) Find the level of each vertex in the tree to the right.
(*ii*) What is the height of the tree?



**Solution**:
(*i*) The root $a$ is at level 0. Vertices $b$, $j$, and $k$ are at level 1. Vertices $c$, $e$, $f$, and $l$ are at level 2. Vertices $d$, $g$, $i$, $m$, and $n$ are at level 3. Vertex $h$ is at level 4.
(*ii*) The height is 4, since 4 is the largest level of any vertex.

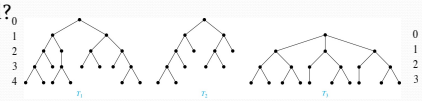
58

## Balanced *m*-Ary Trees

**Definition**: A rooted $m$-ary tree of height $h$ is *balanced* if all leaves are at levels $h$ or $h-1$.

**Example**: Which of the rooted trees shown below is balanced?

Levels




**Solution**: $T_1$ and $T_3$ are balanced, but $T_2$ is not because it has leaves at levels 2, 3, and 4.

59

## The Bound for the Number of Leaves in an *m*-Ary Tree

**Theorem 5**: There are at most $m^h$ leaves in an $m$-ary tree of height $h$.

***Proof* (*by mathematical induction on height*)**:

*BASIS STEP*: Consider an $m$-ary trees of height 1. The tree consists of a root and no more than $m$ children, all leaves. Hence, there are no more than $m^1 = m$ leaves in an $m$-ary tree of height 1.

*INDUCTIVE STEP*: Assume the result is true for all $m$-ary trees of height $< h$. Let $T$ be an $m$-ary tree of height $h$. The leaves of $T$ are the leaves of the subtrees of $T$ we get when we delete the edges from the root to each of the vertices of level 1.


Each of these subtrees has height $\leq h-1$. By the inductive hypothesis, each of these subtrees has at most $m^{h-1}$ leaves. Since there are at most $m$ such subtrees, there are at most $m \cdot m^{h-1} = m^h$ leaves in the tree. ◀

**Corollary 1**: If an $m$-ary tree of height $h$ has $l$ leaves, then $h \geq \lceil \log_m l \rceil$. If the $m$-ary tree is full and balanced, then $h = \lceil \log_m l \rceil$. (*see text for the proof*)

60

# Binary Trees & Applications

61

## Binary Trees and Applications

- The binary tree is a fundamental structure in mathematics and computer science. Some of the terminology of rooted trees, such as, edge, path, branch, leaf, depth, and level number, will also be used for binary trees. However, we will use the term node, rather than vertex, with binary trees. We emphasize that a binary tree is *not* a special case of a rooted tree; they are different mathematical objects.

62

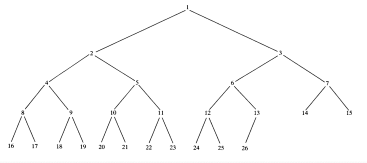
## Binary Trees and Applications

- 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. This 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.

63

## Binary Trees and Applications

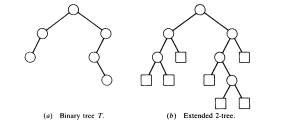
- **Complete and Extended Binary trees**: a binary tree is *complete* if at all levels, except possibly the last have the maximum number of possible nodes and if all the nodes on the last level appear as far left as possible. The diagram shows a complete binary tree of 26 nodes:



64

## Binary Trees and Applications

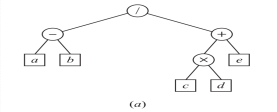
- Extended Binary Trees—2 Trees
- A binary tree tree T is said to be a 2-tree or an extended binary tree if each node N has either 0 or 2 children. In such a case, the nodes with two children are called internal nodes, and the nodes with 0 children are called external nodes. Sometimes the nodes are distinguished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




65

## Binary Trees and Applications

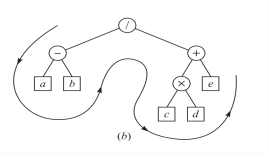
- The Polish mathematician Lukasiewicz noted that if a binary operator is placed *before* the arguments, parentheses are not needed, e.g., +xy rather than x+y or /mn instead of m/n. Suppose F = (a-b)/((c*d)+e). In prefix form F=/-ab+*cde. Using a 2-tree as the data structure and inserting the symbols starting at the root leads to




66

## Binary Trees and Applications

- Which can then be evaluated by starting scanning the tree as shown in this diagram:



- F=/-ab+*cde = (a-b)/((c*d)+e)

67

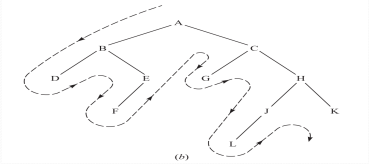
## Binary Trees and Applications

- 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.

68

## Binary Trees and Applications

- Traversals

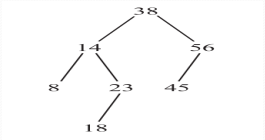


- Preorder Traversal: A B D E F C G H J L K
- Inorder Traversal: D B F E A G C L J H K
- Postorder Traversal: D F E B G L J K H C A
- Note that the terminal nodes are traversed in the same order, left to right,in all three traversals; true for all binary trees

69

## Binary Trees and Applications

- Binary Search Trees:
- Let *T* be a binary tree. *T* is a binary search tree if, for each node *N* in *T*, the value of *N* is greater than every value in its left subtree and is less than every value in the right subtree of *N*.




70

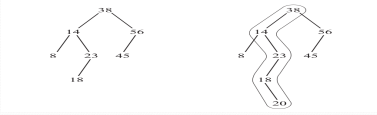
## Binary Trees and Applications

- Searching & Inserting in a Binary Tree
- Given a binary tree *T* and an Object to be found or, if not found, inserted into the tree
  - 1: Compare Object to the root of the tree
    - If Object < N, proceed to the left child of *N*
    - If Object > *N*, proceed to the right child of *N*
  - 2: Repeat 1 until either
    - A node such that *N* = Object is found; search succeeds
    - Arrive at an empty subtree; search fails; insert Object in place of the empty subtree

71

## Binary Trees and Applications

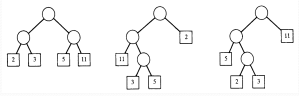
- Consider the binary search tree *T* given below. Suppose Object = 20 is to be found or inserted:
  - Compare Object = 20 with root R=38. Since 20 < 38, proceed to the left child of 38, which is 14.
  - Compare Object = 20 with 14. Since 20 > 14, proceed to the right child of 14, which is 23.
  - Compare Object = 20 with 23. Since 20 < 23, proceed to the left child of 23, which is 18.
  - Compare Object = 20 with 18. Since 20 > 18 and 18 has no right child, insert 20 as the right child of 18.




72

## Binary Trees and Applications

- **Path Lengths, Huffman's Algorithm**
  - Let T be an extended binary tree or 2-tree. Recall that if *T* has *n* external (i.e., leaf) nodes, then *T* has $n - 1$ internal nodes.
- **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 $W_i$ 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 $L_i$.
  - EXAMPLE Given 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.

73

## Binary Trees and Applications

- Huffman's Algorithm
  - The general problem we want to solve is the following. Suppose a list of n weights is given: W1, W2,...,Wn 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 the next slide, 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*.

74

## Binary Trees and Applications

- Huffman's Algorithm:
  - 1: Suppose $n = 1$: Let *T* be the tree with one node, *N*, with weight $w_1$, *then exit.*
  - *2: Suppose* $n > 1$*:*
    - Find two minimum weights $w_i$ and $w_j$ among the given *n* weights
    - Replace $w_i$ and $w_j$ in the list by $w_i + w_j$ so the list has $n - 1$ weights.
    - Find a tree *T'* that gives a minimum weighted path length for the $n-1$ weights.
    - In *T'*, replace the external node $[w_i + w_j]$ by the subtree 
    - Exit



75

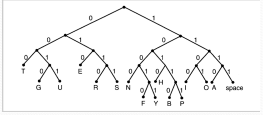
## Binary Trees and Applications

- Example:
- 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 (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 (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)+25(2)+5(4)+2(5)+5(5)+19(3) = 280

76

## Binary Trees and Applications

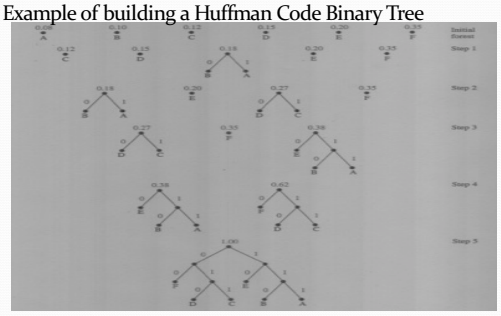
- The binary rooted tree shown in the figure can be used to encode and decode English text according to a Huffman code. The sequence of edges from the root to any letter yields the binary code for that letter. Note that the number of bits varies from one letter to another.
- (a) Decode 00100110 11101011 11101111 10001010 01011010 11010011 11111100 01111111 10010001 11000. The coded message is grouped into strings of eight bits to enhance readability only; the grouping has no other significance.
- (b) Encode 'TO BE OR NOT TO BE'.
- (c) What is the advantage of a Huffman code over codes that use a fixed number of bits for each letter?




77

## Binary Trees and Applications

Example of building a Huffman Code Binary Tree



78

## Decision Trees

- A decision tree is a diagram representation of **possible solutions/consequences** to/of a decision. It shows different outcomes from a set of decisions. The diagram is a widely used **decision-making tool** for analysis and planning.
- **Advantages**
  - A decision tree is easy to understand and interpret.
  - Expert opinion and preferences can be included, as well as hard data.
  - Can be used with other decision techniques.
  - New scenarios can be added easily.
- **Disadvantages**
  - If a decision tree is used for categorical variables with multiple levels, those variables with more levels will have more information gain.
  - Calculations can quickly become very complex (although this is usually only a problem if the tree is being created by hand).

79

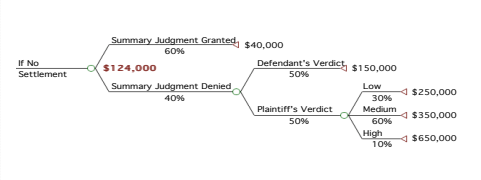
## Categorical Variables

- Categorical variables are those variables that fall into a particular **category**. Hair color, gender, college major, college attended, political affiliation, disability, or sexual orientation are all categories that could have lists of categorical variables. Usually, the variables take on one of a number of fixed values in a set. These variables are not ordered in any sense.
- For example:
  - The category "hair color" could contain the categorical variables "black," "brown," "blonde," and "red."
  - The category "gender" could contain the categorical variables "Male", "Female", or "Other."
- Note that "hair color" and "gender" are the categories and are not categorical variables themselves. A categorical variable is a value that variables in a study take; the value varies from person to person. Let's say you survey people and ask them to tell you their hair color. They would respond with a categorical variable of black, brown, blond, or red. They wouldn't respond "hair color."

80

## A Decision Tree

- The following image, based on an infographic from John DeGroote's website shows how a decision tree can be used to give realistic expectations of what plaintiffs can expect when going to court.




81

# Tree Traversal

Section 11.3

82

## Section Summary

- Traversal Algorithms
- Infix, Prefix, and Postfix Notation

83

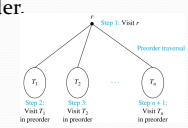
## Tree Traversal

- Procedures for systematically visiting every vertex of an ordered tree are called *traversals*.
- The three most commonly used *traversals* are *preorder traversal*, *inorder traversal*, and *postorder traversal*.

84

## Preorder Traversal

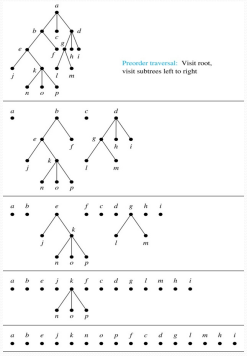
**Definition**: Let $T$ be an ordered rooted tree with root $r$. If $T$ consists only of $r$, then $r$ is the *preorder traversal* of $T$. Otherwise, suppose that $T_1$, $T_2$, ..., $T_n$ are the subtrees of $r$ from left to right in $T$. The preorder traversal begins by visiting $r$, and continues by traversing $T_1$ in preorder, then $T_2$ in preorder, and so on, until $T_n$ is traversed in preorder.




85

## Preorder Traversal (*continued*)

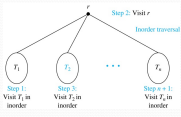
```
procedure preorder (T: ordered rooted tree)
r := root of T
list r
for each child c of r from left to right
   T(c) := subtree with c as root
   preorder(T(c))
```




86

## Inorder Traversal

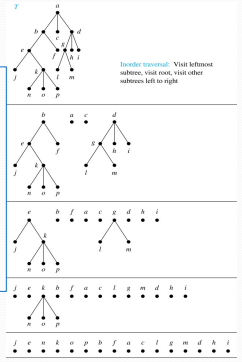
**Definition**: Let $T$ be an ordered rooted tree with root $r$. If $T$ consists only of $r$, then $r$ is the *inorder traversal* of $T$. Otherwise, suppose that $T_1$, $T_2$, ..., $T_n$ are the subtrees of $r$ from left to right in $T$. The inorder traversal begins by traversing $T_1$ in inorder, then visiting $r$, and continues by traversing $T_2$ in inorder, and so on, until $T_n$ is traversed in inorder.




87

## Inorder Traversal (*continued*)

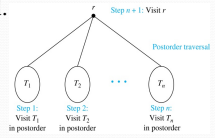
```
procedure inorder (T: ordered rooted tree)
r := root of T
if r is a leaf then list r
else
    l := first child of r from left to right
    T(l) := subtree with l as its root
    inorder(T(l))
    list(r)
    for each child c of r from left to right
        T(c) := subtree with c as root
        inorder(T(c))
```




88

## Postorder Traversal

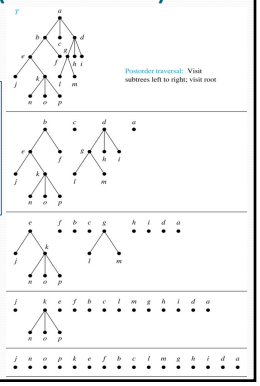
**Definition**: Let $T$ be an ordered rooted tree with root $r$. If $T$ consists only of $r$, then $r$ is the *postorder traversal* of $T$. Otherwise, suppose that $T_1$, $T_2$, ..., $T_n$ are the subtrees of $r$ from left to right in $T$. The postorder traversal begins by traversing $T_1$ in postorder, then $T_2$ in postorder, and so on, after $T_n$ is traversed in postorder, $r$ is visited.




89

## Postorder Traversal (*continued*)

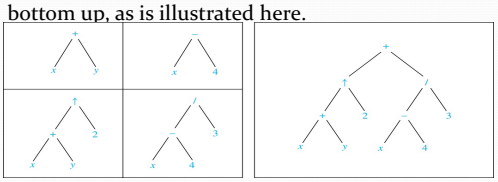
```
procedure postorder (T: ordered rooted tree)
r := root of T
for each child c of r from left to right
    T(c) := subtree with c as root
    postorder(T(c))
list r
```




90

## Expression Trees

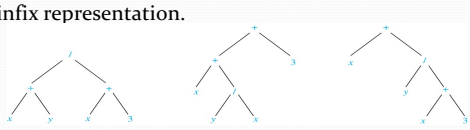
- Complex expressions can be represented using ordered rooted trees.
- Consider the expression $((x + y) \uparrow 2) + ((x - 4)/3)$.
- A binary tree for the expression can be built from the bottom up, as is illustrated here.



91

## Infix Notation

- An inorder traversal of the tree representing an expression produces the original expression when parentheses are included except for unary operations, which now immediately follow their operands.
- We illustrate why parentheses are needed with an example that displays three trees all yield the same infix representation.



92

## Prefix Notation

Jan Łukasiewicz (1878-1956)

- When we traverse the rooted tree representation of an expression in preorder, we obtain the *prefix* form of the expression. Expressions in prefix form are said to be in *Polish notation*, named after the Polish logician Jan Łukasiewicz.
- Operators precede their operands in the prefix form of an expression. Parentheses are not needed as the representation is unambiguous.
- The prefix form of $((x + y) \uparrow 2) + ((x - 4)/3)$ is $+ \uparrow + x\ y\ 2\ /\ -\ x\ 4\ 3$.
- Prefix expressions are evaluated by working from right to left. When we encounter an operator, we perform the corresponding operation with the two operands to the right.

**Example**: We show the steps used to evaluate a particular prefix expression:

+ − * 2 3 5 / ↑ 2 3 4

2 ↑ 3 = 8

+ − * 2 3 5 / 8 4

8 / 4 = 2

+ − * 2 3 5 2

2 * 3 = 6

+ − 6 5 2

6 − 5 = 1

+ 1 2

1 + 2 = 3

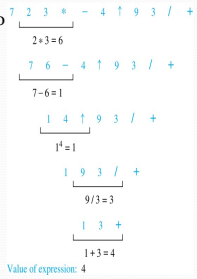
Value of expression: 3

93

## Postfix Notation

- We obtain the *postfix form* of an expression by traversing its binary trees in postorder. Expressions written in postfix form are said to be in *reverse Polish notation.*
- Parentheses are not needed as the postfix form is unambiguous.
- $x\ y + 2 \uparrow x\ 4 - 3 / +$ is the postfix form of $((x + y) \uparrow 2) + ((x - 4)/3)$.
- A binary operator follows its two operands. So, to evaluate an expression one works from left to right, carrying out an operation represented by an operator on its preceding operands.

**Example**: We show the steps used to evaluate a particular postfix expression.




94

## Spanning Trees

Section 11.4/11.5

95

## Section Summary

- Spanning Trees
- Depth-First Search
- Breadth-First Search
- Depth-First Search in Directed Graphs
- Minimal Spanning Trees and Algorithms

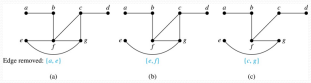
96

# Spanning Trees

**Definition**: Let $G$ be a simple graph. A spanning tree of $G$ is a subgraph of $G$ that is a tree containing every vertex of $G$.

**Example**: Find the spanning tree of this graph:

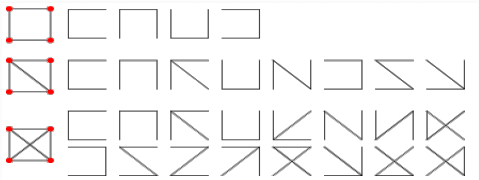
**Solution**: The graph is connected, but is not a tree because it contains simple circuits. Remove the edge $\{a, e\}$. Now one simple circuit is gone, but the remaining subgraph still has a simple circuit. Remove the edge $\{e, f\}$ and then the edge $\{c, g\}$ to produce a simple graph with no simple circuits. It is a spanning tree, because it contains every vertex of the original graph.




97

# Spanning Trees

- Spanning tree examples (compliments of Wolfram):



- These graphs make the obvious point that the spanning tree of a simple graph is not unique. However, there are times that one may wish to pick a "best" spanning tree in terms of some specific criteria—usually a "minimal" spanning tree; i.e. a tree chosen to minimize some specific criterion, say the lengths of interconnections in a network…

98

# Spanning Trees (*continued*)

**Theorem**: A simple graph is connected if and only if it has a spanning tree.

***Proof***: Suppose that a simple graph $G$ has a spanning tree $T$. $T$ contains every vertex of $G$ and there is a path in $T$ between any two of its vertices. Because $T$ is a subgraph of $G$, there is a path in $G$ between any two of its vertices. Hence, $G$ is connected.

Now suppose that $G$ is connected. If $G$ is not a tree, it contains a simple circuit. Remove an edge from one of the simple circuits. The resulting subgraph is still connected because any vertices connected via a path containing the removed edge are still connected via a path with the remaining part of the simple circuit. Continue in this fashion until there are no more simple circuits. A tree is produced because the graph remains connected as edges are removed. The resulting tree is a spanning tree because it contains every vertex of $G$.

99

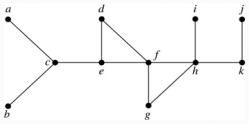
## Depth-First Search

- To use *depth-first search* to build a spanning tree for a connected simple graph first arbitrarily choose a vertex of the graph as the root.
  - Form a path starting at this vertex by successively adding vertices and edges, where each new edge is incident with the last vertex in the path and a vertex not already in the path. Continue adding vertices and edges to this path as long as possible.
  - If the path goes through all vertices of the graph, the tree consisting of this path is a spanning tree.
  - Otherwise, move back to the next to the last vertex in the path, and if possible, form a new path starting at this vertex and passing through vertices not already visited. If this cannot be done, move back another vertex in the path.
  - Repeat this procedure until all vertices are included in the spanning tree.

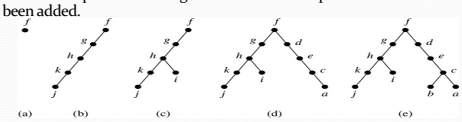
100

## Depth-First Search (*continued*)

**Example**: Use depth-first search to find a spanning tree of this graph.



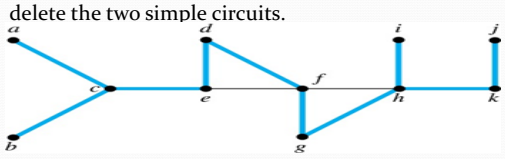

**Solution**: We start arbitrarily with vertex *f*. We build a path by successively adding an edge that connects the last vertex added to the path and a vertex not already in the path, as long as this is possible. The result is a path that connects *f*, *g*, *h*, *k*, and *j*. Next, we return to *k*, but find no new vertices to add. So, we return to *h* and add the path with one edge that connects *h* and *i*. We next return to *f*, and add the path connecting *f*, *d*, *e*, *c*, and *a*. Finally, we return to *c* and add the path connecting *c* and *b*. We now stop because all vertices have been added.




101

## Depth-First Search (*continued*)

- The edges selected by depth-first search of a graph are called *tree edges*. All other edges of the graph must connect a vertex to an ancestor or descendant of the vertex in the graph. These are called *back edges*.
- In this figure, the tree edges are shown with heavy blue lines. The two thin black edges are back edges; note that these are the two edges that must be removed to delete the two simple circuits.




102

## Depth-First Search Algorithm

- We now use pseudocode to specify depth-first search. In this recursive algorithm, after adding an edge connecting a vertex $v$ to the vertex $w$, we finish exploring $w$ before we return to $v$ to continue exploring from $v$.

```
procedure DFS(G: connected graph with vertices v1, v2, ..., vn)
T := tree consisting only of the vertex v1
visit(v1)

procedure visit(v: vertex of G)
for each vertex w adjacent to v and not yet in T
   add vertex w and edge {v,w} to T
   visit(w)
```

103

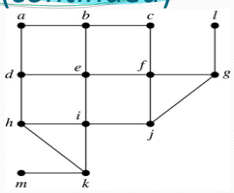
## Breadth-First Search

- We can construct a spanning tree using *breadth-first search*. We first arbitrarily choose a root from the vertices of the graph.
  - Then we add all of the edges incident to this vertex and the other endpoint of each of these edges. We say that these are the vertices at level 1.
  - For each vertex added at the previous level, we add each edge incident to this vertex, *as long as it does not produce a simple circuit*. The new vertices we find are the vertices at the next level.
  - We continue in this manner until all the vertices have been added and we have a spanning tree.

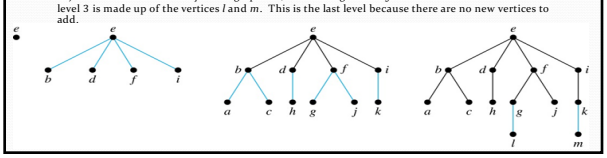
104

## Breadth-First Search (*continued*)

**Example**: Use breadth-first search to find a spanning tree for this graph.




**Solution**: We arbitrarily choose vertex $e$ as the root. We then add the edges from $e$ to $b$, $d$, $f$, and $i$. These four vertices make up level 1 in the tree. Next, we add the edges from $b$ to $a$ and $c$, the edges from $d$ to $h$ *(only to $h$ as $a$ is already in the graph)*, the edges from $f$ to $j$ and $g$, and the edge from $i$ to $k$. The endpoints of these edges not at level 1 are at level 2. Next, add edges from these vertices to adjacent vertices not already in the graph. So, we add edges from $g$ to $l$ and from $k$ to $m$. We see that level 3 is made up of the vertices $l$ and $m$. This is the last level because there are no new vertices to add.




105

# Breadth-First Search Algorithm

- A pseudocode description of the breadth-first search:

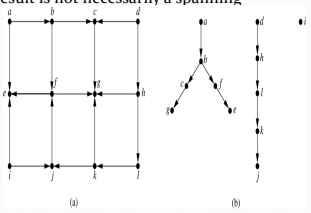
```
procedure BFS(G: connected graph with vertices v1, v2, ..., vn)
T := tree consisting only of the vertex v1
L := empty list
put v1 in the list L of unprocessed vertices
while L is not empty
   remove the first vertex, v, from L
   for each neighbor w of v
      if w is not in L and not in T then
         add  w to the end of the list L
         add  w and edge {v,w} to T
```

106

# Depth-First Search in Directed Graphs

- Both depth-first search and breadth-first search can be easily modified to run on a directed graph. But the result is not necessarily a spanning tree, but rather a spanning forest.

**Example**: For the graph in (a), if we begin at vertex *a*, depth-first search adds the path connecting *a*, *b*, *c*, and *g*. At *g*, we are blocked, so we return to *c* which is blocked so return to *b*. Next, we add the paths connecting *b* to *f* and *f* to *e*. Next, we return to *a* and find that we cannot add a new path. So, we begin another tree with *d* as its root. We find that this new tree consists of the path connecting the vertices *d*, *h*, *l*, *k*, and *j*. Finally, we add a new tree, which only contains *i*, its root.



- To index websites, search engines such as Google systematically explore the web starting at known sites. The programs that do this exploration are known as *Web spiders*. They may use both breath-first search or depth-first search to explore the Web graph.

107

## How to find minimum spanning tree?

- The **stupid method** is to list all spanning trees and find the minimum of the list. We already know how to find minima…but in practice there are far too many trees for this to be efficient. It's also not really an algorithm, because you'd still need to know how to list all the trees. A better idea is to find some key property of the MST that lets us be sure that some edge is part of it, and use this property to build up the MST one edge at a time.
- For simplicity, we assume that there is a unique minimum spanning tree (not always true, but this *is* a starting point). You can get ideas like this to work without this assumption but it becomes harder to state your theorems or write your algorithms precisely.

- **Lemma:** Let X be any subset of the vertices of G, and let edge e be the smallest edge connecting X to G-X. Then e is part of the minimum spanning tree.
- Proof (by contradiction): Suppose you have a tree T not containing e; then we want to show that T is not the MST. Let e=(u,v), with u in X and v not in X. Then because T is a spanning tree it contains a unique path from u to v, which together with e forms a cycle in G. This path has to include another edge *f* connecting X to G-X. T+e-*f* is another spanning tree (it has the same number of edges, and remains connected since you can replace any path containing *f* by one going the other way around the cycle). It has smaller weight than T since e has smaller weight than *f*. So T was not minimum, which is what we wanted to prove.

108

## How To Find A Minimum Spanning Tree

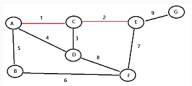
- **Kruskal's algorithm**
- We'll start with Kruskal's algorithm, which is easiest to understand and probably the best one for solving problems by hand.
- **Kruskal's algorithm**:
  1. sort the edges of G in increasing order by length
  2. keep a subgraph S of G, initially empty
  3. for each edge e in sorted order
     1. if the endpoints of e are disconnected in S add e to S
  4. return S
- This algorithm is known as a *greedy algorithm*, because it chooses at each step the cheapest edge to add to S. You should be very careful when trying to use greedy algorithms to solve other problems, since it usually doesn't work; e.g., if you want to find a shortest path from a to b, it might be a bad idea to keep taking the shortest edges. The greedy idea only works in Kruskal's algorithm because of the key property we proved.

109

## Kruskal's Algorithm

- Find the edge of lowest weight and choose it:

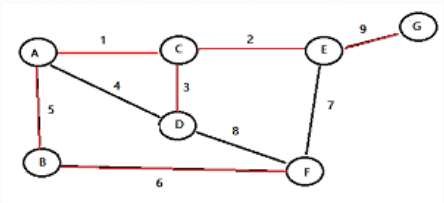
- Continue selecting edges of lowest remaining weight until all nodes are in the tree:



- Note: if you have multiple edges of the same weight, just pick one…
- Note: at each step, check to see if the edge added forms a cycle; if so, discard the choice and try the edge of next smallest weight

110

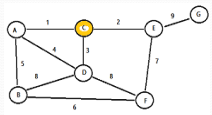
## The MST From This Example



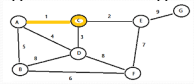

111

# Prim's Algorithm

- Choose a random node:



- Find all of the edges that go to un-highlighted nodes, and choose the edge of lowest weight and highlight that node:



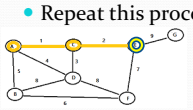
- Repeat this process until the MST is found:

The next lowest weight edge is (C,E) of wt. 2; select it and add node E to the tree.

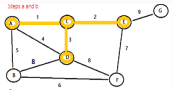
112

# Prim's Algorithm

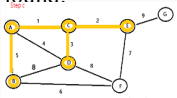
- Repeat this process until the MST is found:



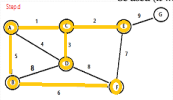
The lowest weight remaining edge is (C,D) of wt 3; select it and add D to the tree



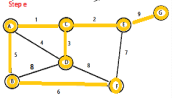
The edge (A,D) of wt. 4 cannot be used (it would form a cycle)



The next selectable edge is (A,B) of wt 5; add B to the tree



The next selectable edge is of wt. 6: (B, F); add F to the tree



The next edge is (F,E) of wt. 7 but that would give a cycle as would both edges of wt. 8; so, select edge (E,G) of wt. 9; add G to the tree. As all nodes are now in the tree, we have found a minimal spanning tree of total weight 26.

Thanks to: https://www.statisticshowto.datasciencecentral.com/minimum-spanning-tree/

113