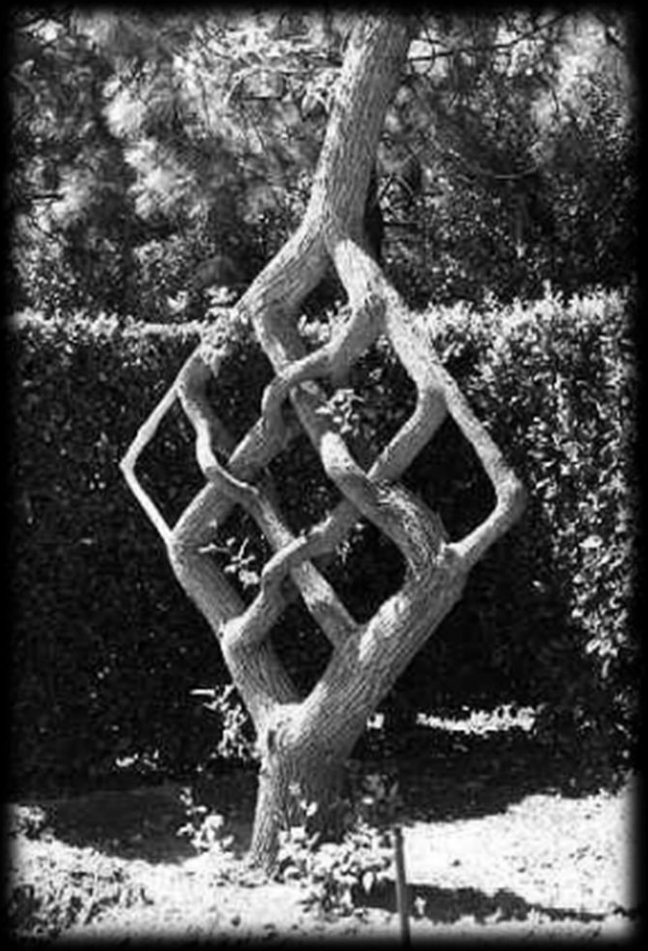
INTRODUCTION TO TREES



COMP 251

Topic Goals

- To learn the basic definitions and properties of trees.

Topic Goals

- To learn the basic definitions and properties of trees.
- To understand what properties of a relation are required in order to define a tree.

Topic Goals

- To learn the basic definitions and properties of trees.
- To understand what properties of a relation are required for it to define a tree.
- To learn the basic formulae for counting internal vertices, leaves, and edges of a tree.

Topic Goals

- To learn the basic definitions and properties of trees.
- To understand what properties of a relation are required for it to define a tree.
- To learn the basic formulae for counting internal vertices, leaves, and edges of a tree.
- To learn the basic formula for determining the height/depth of a tree.

Topic Goals

- To learn the basic definitions and properties of trees.
- To understand what properties of a relation are required for it to define a tree.
- To learn the basic formulae for counting internal vertices, leaves, and edges of a tree.
- To learn the basic formula for determining the height/depth of a tree.
- To develop skill at representing trees as graphical objects and as ordered triples.

Topic Goals

- To learn the basic definitions and properties of trees.
- To understand what properties of a relation are required for it to define a tree.
- To learn the basic formulae for counting internal vertices, leaves, and edges of a tree.
- To learn the basic formula for determining the height/depth of a tree.
- To develop skill at representing trees as graphical objects and as ordered triples.
- To discuss graphs and trees as data structures.

Recall: What is Graph Theory?

- The study of graphs.
  - Informally, graphs are mathematical structures used to model binary relations between elements of a given set.

Recall: What is Graph Theory?

- The study of graphs.
  - Informally, graphs are mathematical structures used to model binary relations between elements of a given set.
- The terminology of Graph Theory is nicely summarized at Wikipedia

  http://en.wikipedia.org/wiki/Glossary_of_graph_theory

Recall: What is Graph Theory?

- The study of graphs.
  - Informally, graphs are mathematical structures used to model binary relations between elements of a given set.
- The terminology of Graph Theory is nicely summarized at Wikipedia

  http://en.wikipedia.org/wiki/Glossary_of_graph_theory
- The origin of the theory is credited to Euler, 1736 – The Königsberg Bridge Problem.

Definition 11.1 Let V be a finite nonempty set, and let $E \subseteq V \times V$. The pair (V, E) is then called a *directed graph* (on V), or *digraph*† (on V), where V is the set of *vertices*, or *nodes*, and E is its set of *edges*. We write $G = (V, E)$ to denote such a digraph.

Definition 11.1 Let V be a finite nonempty set, and let $E \subseteq V \times V$. The pair (V, E) is then called a *directed graph* (on V), or *digraph*† (on V), where V is the set of *vertices*, or *nodes*, and E is its set of *edges*. We write $G = (V, E)$ to denote such a digraph.

Figure 11.2 provides an example of a directed graph on $V = \{a, b, c, d, e\}$ with $E = \{(a, a), (a, b), (a, d), (b, c)\}$. The direction of an edge is indicated by placing a directed arrow on the edge, as shown here. For any edge, such as (b, c), we say that the edge is *incident* with the vertices b, c; b is said to be *adjacent to* c, whereas c is *adjacent from* b. In addition, vertex b is called the *origin*, or *source*, of the edge (b, c), and vertex c is the *terminus*, or *terminating vertex*. The edge (a, a) is an example of a *loop*, and the vertex e that has no incident edges is called an *isolated* vertex.

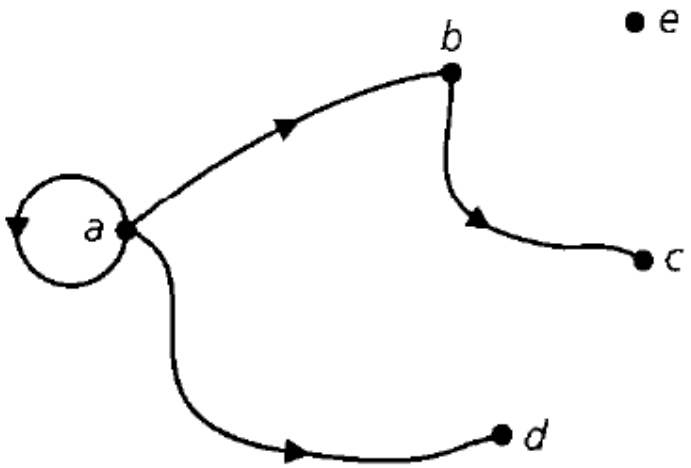


Figure 11.2

When there is no concern about the direction of any edge, the structure $G = (V, E)$, where E is now a set of unordered pairs from V, is called an *undirected graph*. An undirected graph is shown in Fig. 11.3(a). This graph is a more compact way of describing the directed graph given in Fig. 11.3(b). In an undirected graph, there are undirected edges such as $\{a, b\}, \{b, c\}, \{a, c\}, \{c, d\}$ in Fig. 11.3(a). An edge such as $\{a, b\}$ stands for $\{(a, b), (b, a)\}$. Although $(a, b) = (b, a)$ only when $a = b$, we do have $\{a, b\} = \{b, a\}$ for any a, b. We can write $\{a, a\}$ to denote a loop in an undirected graph, but $\{a, a\}$ is considered the same as (a, a).

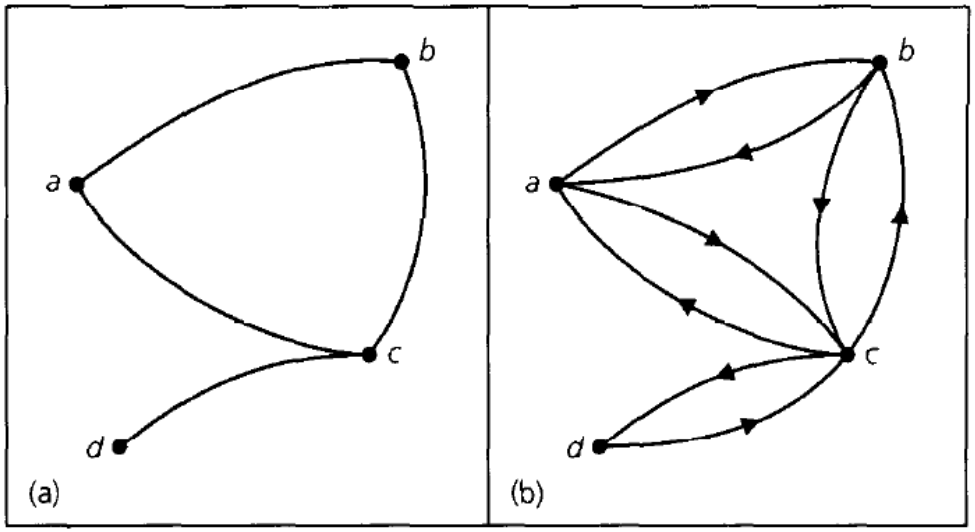
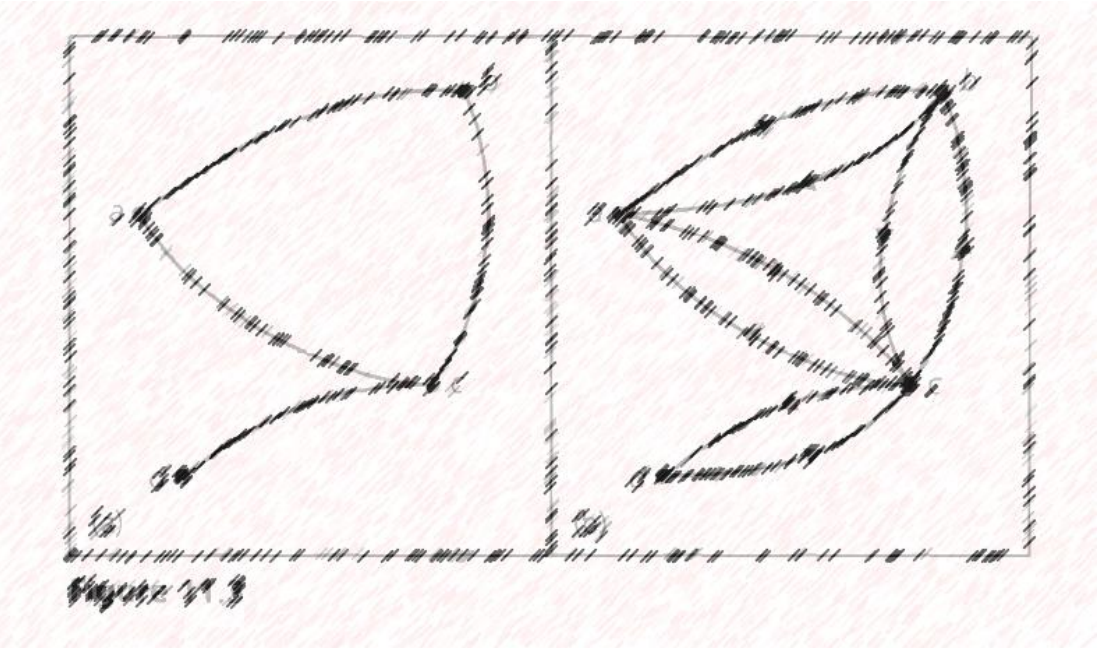


Figure 11.3

In general, if a graph G is not specified as directed or undirected, it is assumed to be undirected. When it contains no loops it is called *loop-free*.



Grimaldi

Basic Terminology

A **graph** $G(V, E)$ is a set V of **vertices** and a set E of **edges**. In an **undirected** graph, an edge is an unordered pair of vertices. An ordered pair of vertices is called a **directed** edge. If we allow **multi-sets** of edges, i.e. multiple edges between two vertices, we obtain a **multigraph**. A self-loop or **loop** is an edge between a vertex and itself. An undirected graph without loops or multiple edges is known as a **simple** graph. In this class we will assume graphs to be simple unless otherwise stated.

https://stanford.edu/~rezab/discrete/Notes/2.pdf

Basic Terminology

A **graph** $G(V, E)$ is a set V of **vertices** and a set E of **edges**. In an **undirected** graph, an edge is an unordered pair of vertices. An ordered pair of vertices is called a **directed** edge. If we allow **multi-sets** of edges, i.e. multiple edges between two vertices, we obtain a **multigraph**. A self-loop or **loop** is an edge between a vertex and itself. An undirected graph without loops or multiple edges is known as a **simple** graph. In this class we will assume graphs to be simple unless otherwise stated.

If vertices a and b are endpoints of an edge, we say that they are **adjacent** and write $a \sim b$. If vertex a is one of edge e's endpoints, a is **incident** to e and we write $a \in e$. The **degree** of a vertex is the number of edges incident to it.

https://stanford.edu/~rezab/discrete/Notes/2.pdf

Basic Terminology

A **graph** $G(V, E)$ is a set V of **vertices** and a set E of **edges**. In an **undirected** graph, an edge is an unordered pair of vertices. An ordered pair of vertices is called a **directed** edge. If we allow **multi-sets** of edges, i.e. multiple edges between two vertices, we obtain a **multigraph**. A self-loop or **loop** is an edge between a vertex and itself. An undirected graph without loops or multiple edges is known as a **simple** graph. In this class we will assume graphs to be simple unless otherwise stated.

If vertices a and b are endpoints of an edge, we say that they are **adjacent** and write $a \sim b$. If vertex a is one of edge e's endpoints, a is **incident** to e and we write $a \in e$. The **degree** of a vertex is the number of edges incident to it.

A **walk** is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $\forall i \in 1, 2, \ldots, k-1$, $v_i \sim v_{i+1}$. A **path** is a walk where $v_i \neq v_j$, $\forall i \neq j$. In other words, a path is a walk that visits each vertex at most once. A **closed walk** is a walk where $v_1 = v_k$. A **cycle** is a closed path, i.e. a path combined with the edge (v_k, v_1). A graph is **connected** if there exists a path between each pair of vertices. A **tree** is a connected graph with no cycles. A **forest** is a graph where each connected component is a tree. A node in a forest with degree 1 is called a **leaf**.

https://stanford.edu/~rezab/discrete/Notes/2.pdf

Basic Terminology

A **graph** $G(V, E)$ is a set V of **vertices** and a set E of **edges**. In an **undirected** graph, an edge is an unordered pair of vertices. An ordered pair of vertices is called a **directed** edge. If we allow **multi-sets** of edges, i.e. multiple edges between two vertices, we obtain a **multigraph**. A self-loop or **loop** is an edge between a vertex and itself. An undirected graph without loops or multiple edges is known as a **simple** graph. In this class we will assume graphs to be simple unless otherwise stated.

If vertices a and b are endpoints of an edge, we say that they are **adjacent** and write $a \sim b$. If vertex a is one of edge e's endpoints, a is **incident** to e and we write $a \in e$. The **degree** of a vertex is the number of edges incident to it.

A **walk** is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $\forall i \in 1, 2, \ldots, k-1$, $v_i \sim v_{i+1}$. A **path** is a walk where $v_i \neq v_j$, $\forall i \neq j$. In other words, a path is a walk that visits each vertex at most once. A **closed walk** is a walk where $v_1 = v_k$. A **cycle** is a closed path, i.e. a path combined with the edge (v_k, v_1). A graph is **connected** if there exists a path between each pair of vertices. A **tree** is a connected graph with no cycles. A **forest** is a graph where each connected component is a tree. A node in a forest with degree 1 is called a **leaf**.

The size of a graph is the number of vertices of that graph. We usually denote the number of vertices with n and the number edges with m.

https://stanford.edu/~rezab/discrete/Notes/2.pdf

Basic Terminology

Definition 11.2 Let x, y be (not necessarily distinct) vertices in an undirected graph $G = (V, E)$. An *x-y walk* in G is a (loop-free) finite alternating sequence

$$x = x_0, e_1, x_1, e_2, x_2, e_3, \ldots, e_{n-1}, x_{n-1}, e_n, x_n = y$$

Grimaldi

Basic Terminology

Definition 11.2 Let x, y be (not necessarily distinct) vertices in an undirected graph $G = (V, E)$. An *x-y walk* in G is a (loop-free) finite alternating sequence

$$x = x_0, e_1, x_1, e_2, x_2, e_3, \ldots, e_{n-1}, x_{n-1}, e_n, x_n = y$$

Definition 11.3 Consider any x-y walk in an undirected graph $G = (V, E)$.

a) If no edge in the x-y walk is repeated, then the walk is called an *x-y trail.* A closed x-x trail is called a *circuit.*

b) When no vertex of the x-y walk occurs more than once, the walk is called an *x-y path.* The term *cycle* is used to describe a closed x-x path.

Grimaldi

Basic Terminology

Definition 11.2 Let x, y be (not necessarily distinct) vertices in an undirected graph $G = (V, E)$. An *x-y walk* in G is a (loop-free) finite alternating sequence

$$x = x_0, e_1, x_1, e_2, x_2, e_3, \ldots, e_{n-1}, x_{n-1}, e_n, x_n = y$$

Definition 11.3 Consider any x-y walk in an undirected graph $G = (V, E)$.

a) If no edge in the x-y walk is repeated, then the walk is called an *x-y trail.* A closed x-x trail is called a *circuit.*

b) When no vertex of the x-y walk occurs more than once, the walk is called an *x-y path.* The term *cycle* is used to describe a closed x-x path.

Convention: In dealing with circuits, we shall always understand the presence of at least one edge. When there is only one edge, then the circuit is a loop (and the graph is no longer loop-free). Circuits with two edges arise in multigraphs, a concept we shall define shortly.

The term *cycle* will always imply the presence of at least three distinct edges (from the graph).

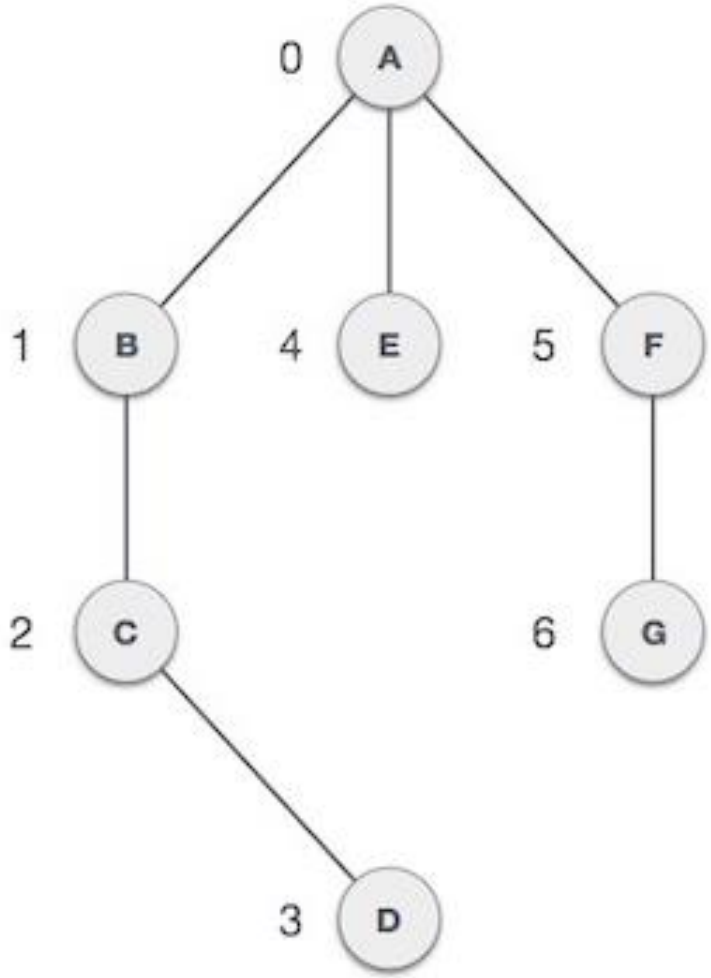
Grimaldi

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms:

- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

Graph Data Structure



Graph Data Structure

The following are the basic primary operations of a Graph:

- **Add Vertex** – Adds a vertex to the graph.
- **Add Edge** – Adds an edge between the two vertices of the graph.
- **Display Vertex** – Displays a vertex of the graph.

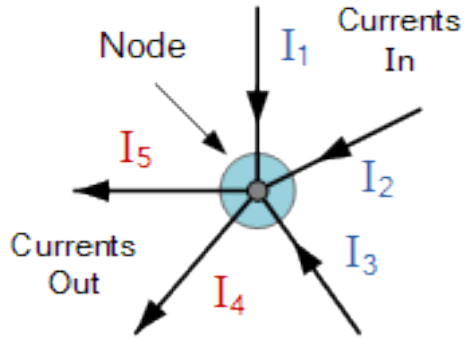

Trees

Trees

Continuing our study of graph theory, we shall now focus on a special type of graph called a tree. First used in 1847 by Gustav Kirchhoff (1824–1887) in his work on electrical networks, trees were later redeveloped and named by Arthur Cayley (1821–1895). In 1857 Cayley used these special graphs in order to enumerate the different isomers of the saturated hydrocarbons $C_n H_{2n+2}, n \in \mathbf{Z}^+$.

With the advent of digital computers, many new applications were found for trees. Special types of trees are prominent in the study of data structures, sorting, and coding theory, and in the solution of certain optimization problems.

Currents Entering the Node
Equals
Currents Leaving the Node



$$I_1 + I_2 + I_3 + (-I_4 + -I_5) = 0$$

Recall the Game Trees of Computational Complexity

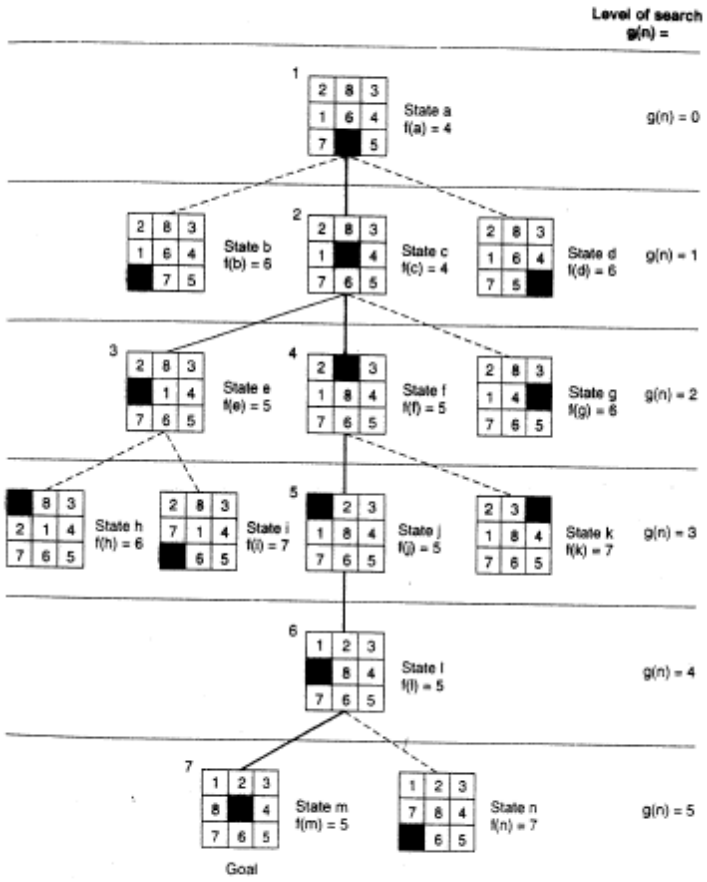
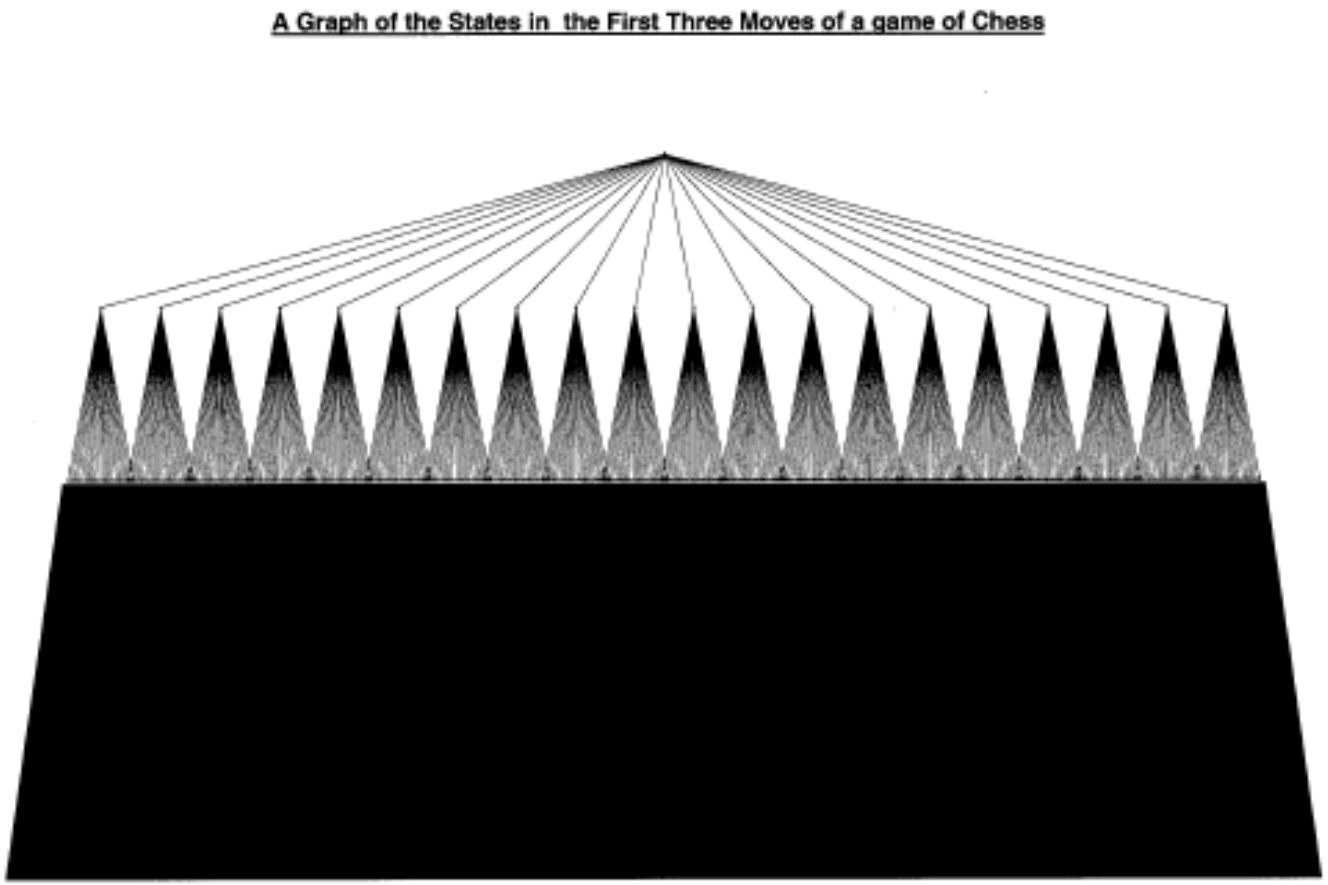


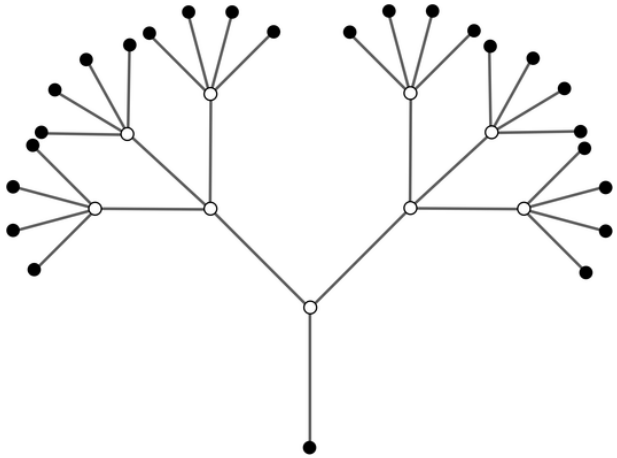
Figure 4.10 State space generated in heuristic search of the 8-puzzle graph.

Recall the Game Trees of Computational Complexity



Trees

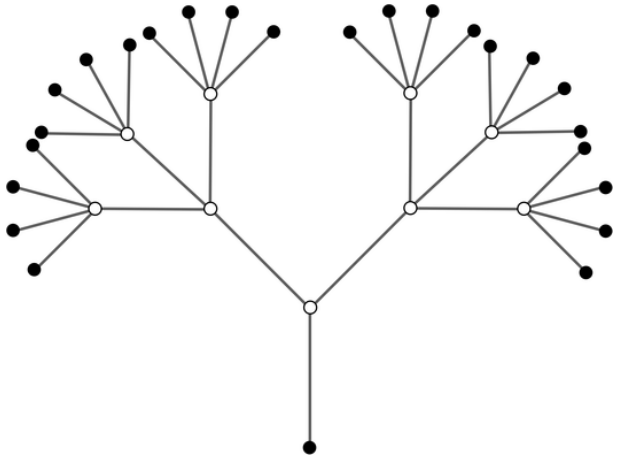
In mathematics, and more specifically in graph theory, a **tree** is an undirected graph in which any two vertices are connected by *exactly one* simple path. In other words, any connected graph without simple cycles is a tree.

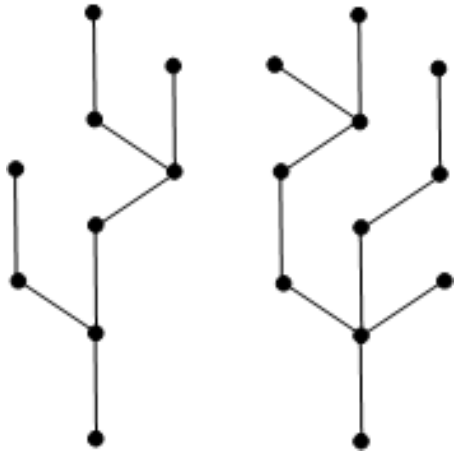


Wikipedia

Trees

In mathematics, and more specifically in graph theory, a **tree** is an undirected graph in which any two vertices are connected by *exactly one* simple path. In other words, any connected graph without simple cycles is a tree. A **forest** is a disjoint union of trees.



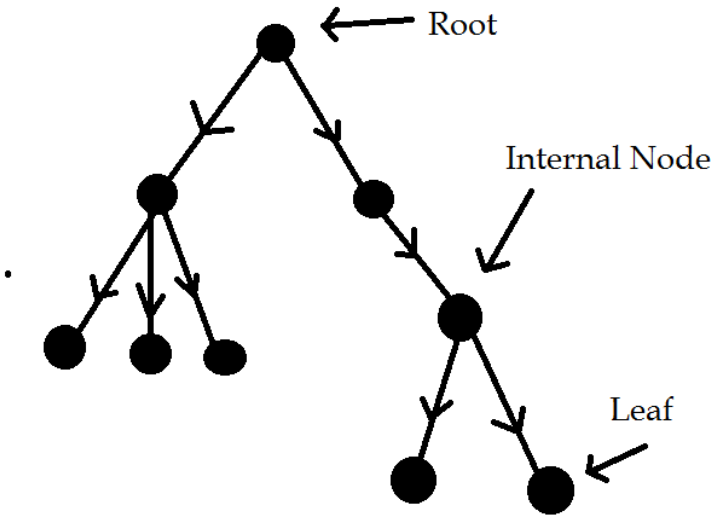


Wikipedia

Trees

In mathematics, and more specifically in graph theory, a **tree** is an undirected graph in which any two vertices are connected by *exactly one* simple path. In other words, any connected graph without simple cycles is a tree. A **forest** is a disjoint union of trees.

The various kinds of data structures referred to as trees in computer science are equivalent as undirected graphs to trees in graph theory, although such data structures are generally **rooted trees**, thus in fact being directed graphs, and may also have additional ordering of branches.



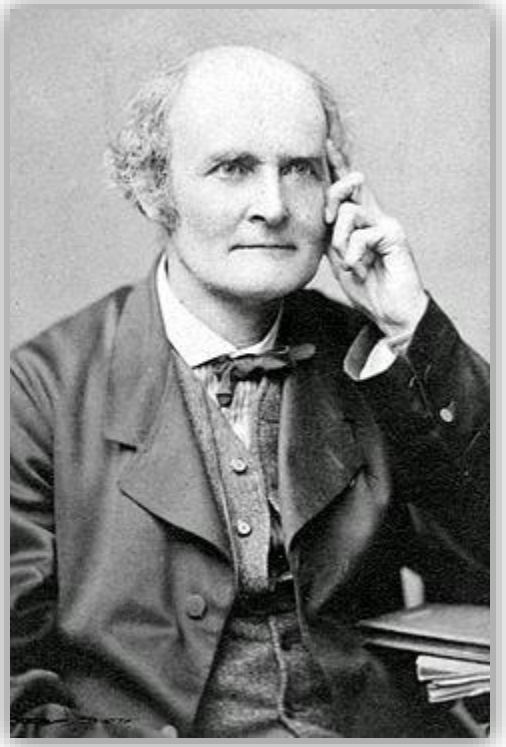
Wikipedia

Trees

In mathematics, and more specifically in graph theory, a **tree** is an undirected graph in which any two vertices are connected by *exactly one* simple path. In other words, any connected graph without simple cycles is a tree. A **forest** is a disjoint union of trees.

The various kinds of data structures referred to as trees in computer science are equivalent as undirected graphs to trees in graph theory, although such data structures are generally **rooted trees**, thus in fact being directed graphs, and may also have additional ordering of branches.

The term "tree" was coined in 1857 by the British mathematician Arthur Cayley.[1]

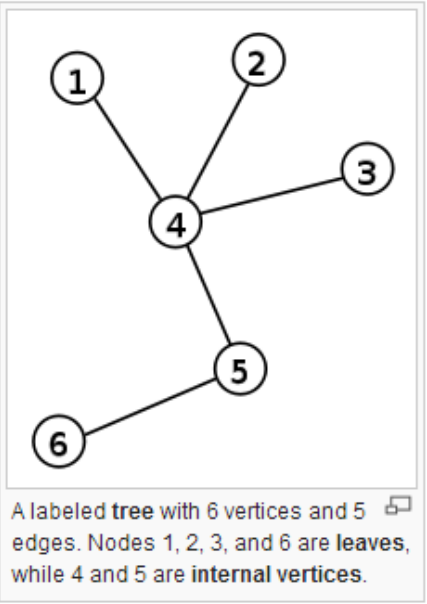


Wikipedia

http://en.wikipedia.org/wiki/**Glossary_of_graph_theory**

Trees [edit]

A **tree** is a connected acyclic simple graph. For directed graphs, each vertex has at most one incoming edge. A vertex of degree 1 is called a **leaf**, or *pendant vertex*. An edge incident to a leaf is a **leaf edge**, or *pendant edge*. (Some people define a leaf edge as a *leaf* and then define a *leaf vertex* on top of it. These two sets of definitions are often used interchangeably.) A non-leaf vertex is an **internal vertex**. Sometimes, one vertex of the tree is distinguished, and called the **root**; in this case, the tree is called **rooted**. Rooted trees are often treated as **directed acyclic graphs** with the edges pointing away from the root.



A labeled **tree** with 6 vertices and 5 edges. Nodes 1, 2, 3, and 6 are **leaves**, while 4 and 5 are **internal vertices**.

A **subtree** of the tree T is a connected subgraph of T.

A **forest** is an acyclic simple graph. For directed graphs, each vertex has at most one incoming edge. (That is, a tree with the connectivity requirement removed; a graph containing multiple disconnected trees.)

A **subforest** of the forest F is a subgraph of F.

A **spanning tree** is a spanning subgraph that is a tree. Every graph has a spanning forest. But only a connected graph has a spanning tree.

A special kind of tree called a **star** is $K_{1,k}$. An induced star with 3 edges is a **claw**.

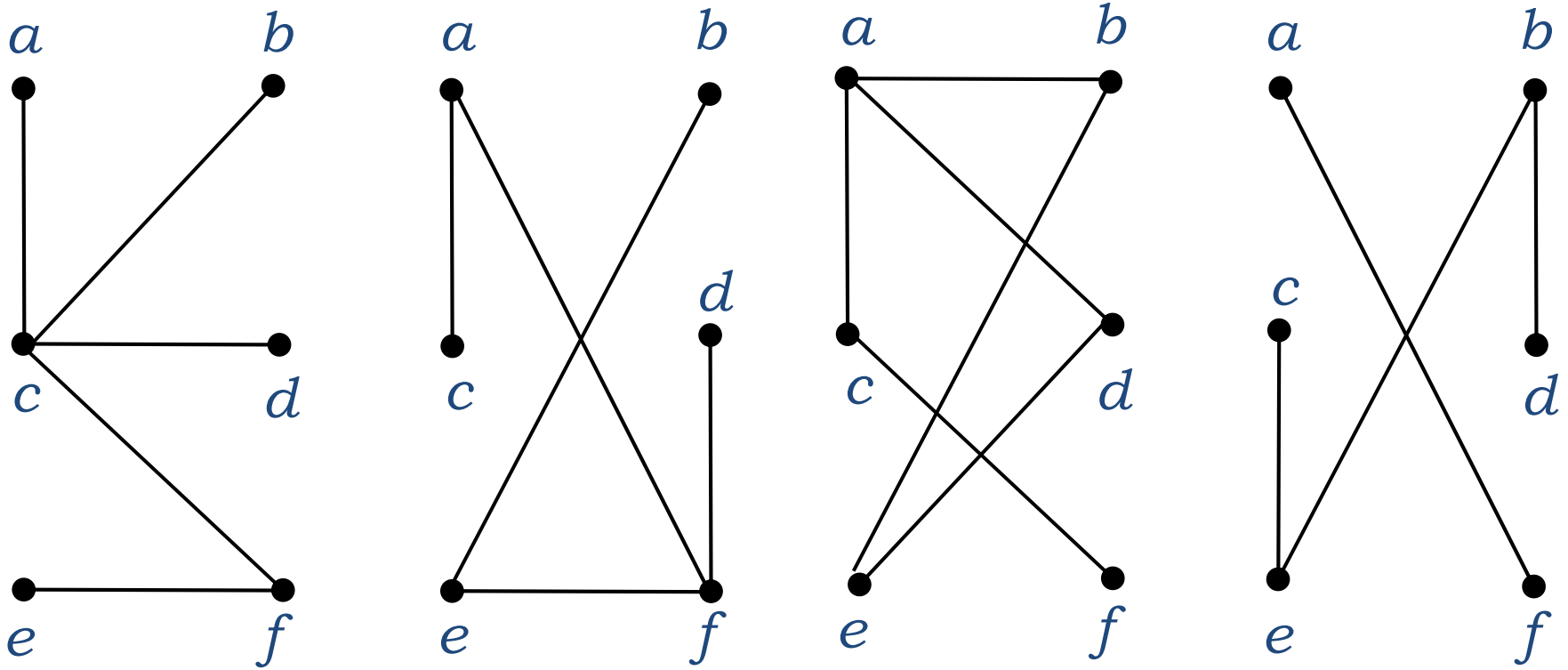
A **caterpillar** is a tree in which all non-leaf nodes form a single path.

A ***k*-ary** tree is a rooted tree in which every internal vertex has no more than k *children*. A 1-ary tree is just a path. A 2-ary tree is also called a **binary tree**.

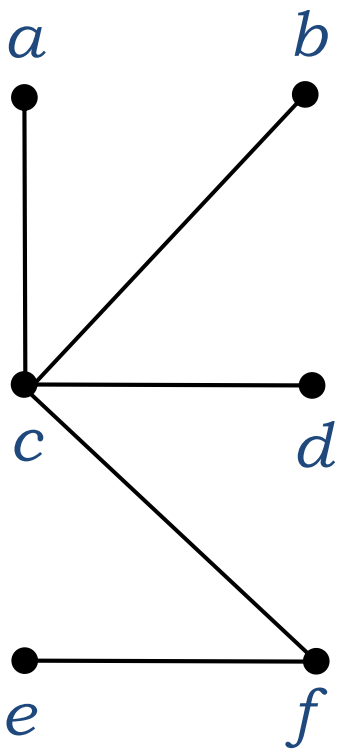
Trees

- A *tree* is a connected undirected graph with
  - No simple circuits
  - No multiple edges
  - No loops
- An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

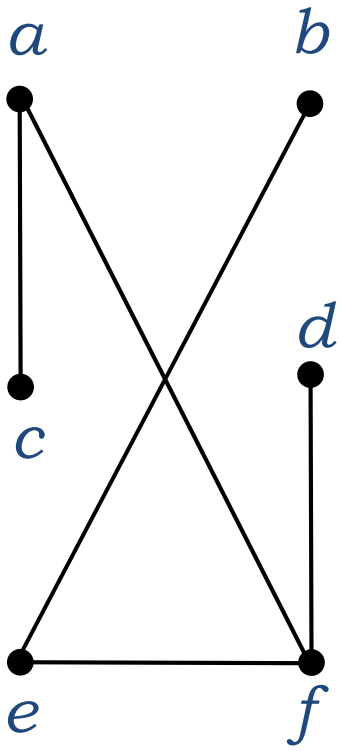
Which graphs are trees?



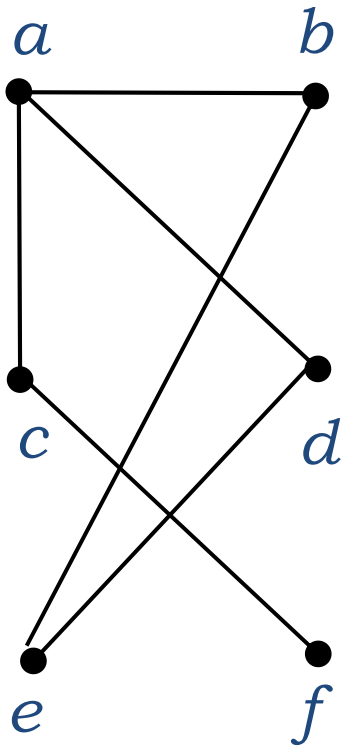
Which graphs are trees?



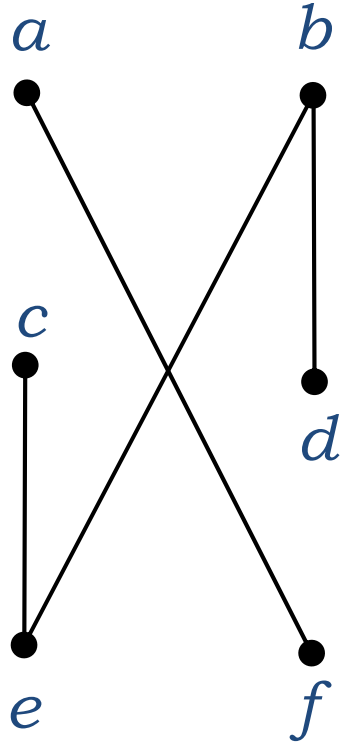
YES



YES



NO

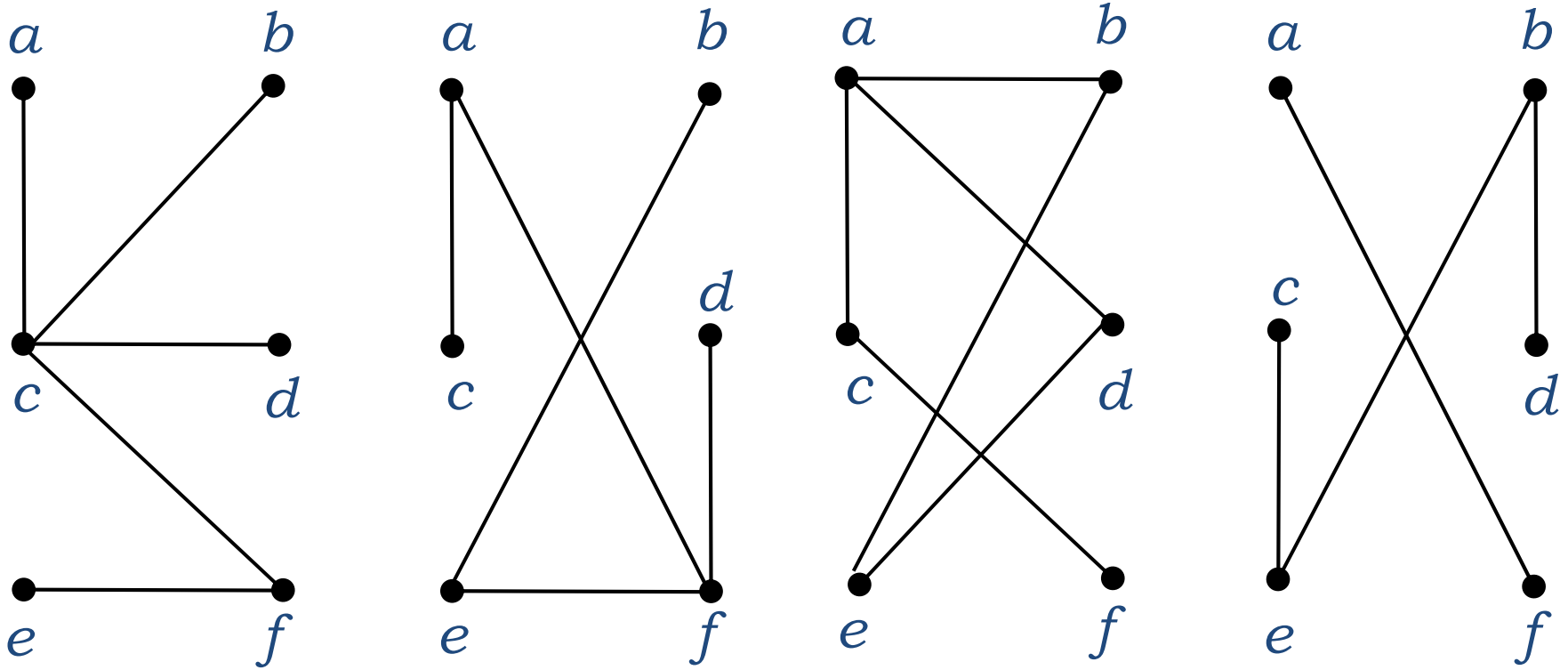


NO

Which graphs are trees?

(e, b, a, d, e) is a simple circuit

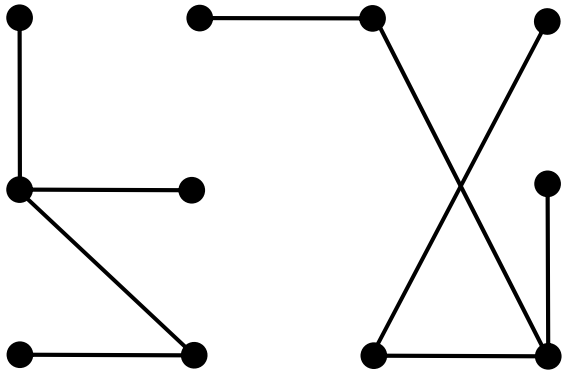
Not connected



YES YES NO NO

Forest

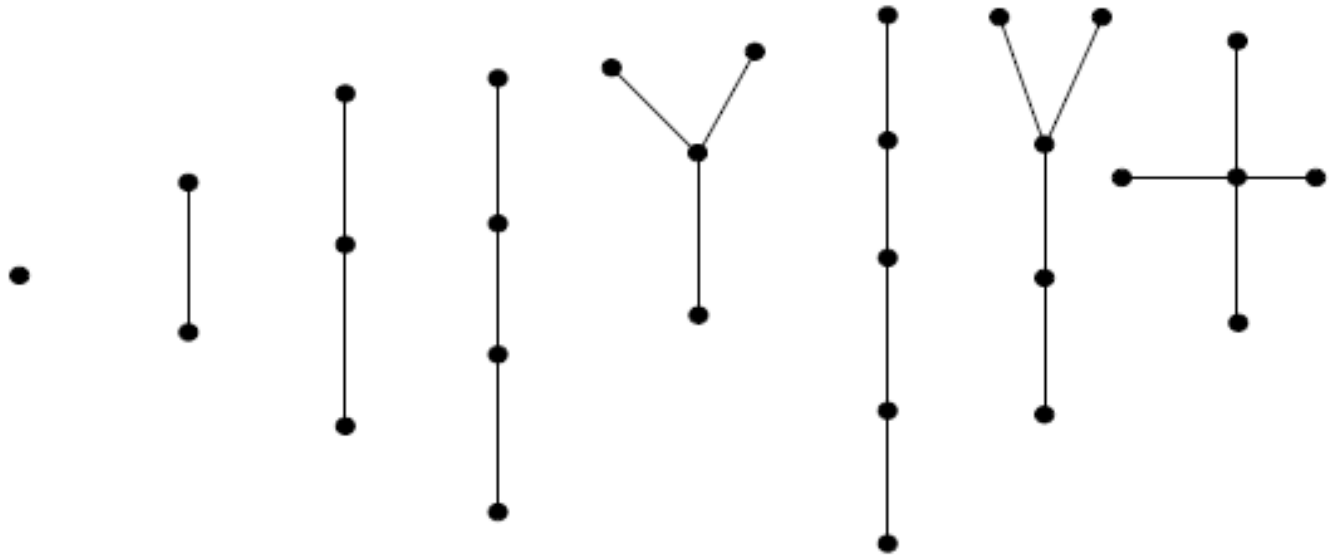
- What if there are no simple circuits but the graph is not connected?
- Each of the connected components is a tree
- The collection is called a *forest*.



Definition: A graph having no cycles is said to be *acyclic*. A *forest* is an acyclic graph.

Definition: A graph having no cycles is said to be *acyclic*. A *forest* is an acyclic graph.

Definition: A *tree* is a connected graph without any cycles, or a tree is a connected acyclic graph. The edges of a tree are called *branches*. It follows immediately from the definition that a tree has to be a simple graph (because self-loops and parallel edges both form cycles).



All trees with fewer than six vertices.

Notes

- A *connected graph* is one in which there exists a path from any node to any other node of said graph.

Notes

- A *connected graph* is one in which there exists a path from any node to any other node of said graph.
- Another way to define a tree is as follows:
  - A graph is a tree *iff* there is exactly one path between every pair of its vertices.

Notes

- A *connected graph* is one in which there exists a path from any node to any other node of said graph.
- Another way to define a tree is as follows:
  - A graph is a tree *iff* there is exactly one path between every pair of its vertices.

Proof:

- If we have a graph *T* which is a tree, then it must be connected with no cycles.
- Since *T* is connected, there must be at least one simple path between each pair of vertices.
- If there is more than one path between two vertices, then parts of those paths could be joined to form a cycle.
- Thus, there must be exactly one path.
- Now suppose we have a graph *G* where there exactly one simple path between vertices.
- This graph is clearly connected.
- If *G* contains a simple circuit, then there are two paths between any vertices on that circuit.
- Thus contradicts the assumption, so there can be no circuits, and *G* is a tree. ■

Another Definition of a Tree

Any connected graph with n vertices and n-1 edges is a tree.

Another Definition of a Tree

Any connected graph with *n* vertices and *n*-1 edges is a tree.

Proof Let G be a connected graph with n vertices and $n-1$ edges. We show that G contains no cycles. Assume to the contrary that G contains cycles.

Remove an edge from a cycle so that the resulting graph is again connected. Continue this process of removing one edge from one cycle at a time till the resulting graph H is a tree. As H has n vertices, so number of edges in H is $n-1$. Now, the number of edges in G is greater than the number of edges in H. So $n-1 > n-1$, which is not possible. Hence, G has no cycles and therefore is a tree. ❑

What is a Tree?

- Informally, a tree is a way to represent graphically certain types of binary relations, a topic introduced earlier. Recall that a binary relation is just a finite subset of pairs of values from a finite set, A, of elements.

What is a Tree?

- Informally, a tree is a way to represent graphically certain types of binary relations, a topic introduced earlier. Recall that a binary relation is just a finite subset of pairs of values from a finite set, A, of elements.
- Earlier a graphical representation of a finite relation $R \subseteq A \times A$ was described, called a digraph. In a digraph each member of the set A is represented by a point or circle in the diagram, called a vertex. As well, every pair in the set R was represented by a directed line or arrow called an edge.

What is a Tree?

- Informally, a tree is a way to represent graphically certain types of binary relations, a topic introduced earlier. Recall that a binary relation is just a finite subset of pairs of values from a finite set, A, of elements.
- Earlier a graphical representation of a finite relation $R \subseteq A \times A$ was described, called a digraph. In a digraph each member of the set A is represented by a point or circle in the diagram, called an vertex. As well, every pair in the set R was represented by a directed line or arrow called an edge.
- Furthermore, some binary relations could be classified according to the existence or absence of certain pairs**. *In particular a tree is a binary relation that is "antireflexive" and symmetric.*** Therefore its digraph will possess **no loops** on any vertex, and **for every edge, (e_i, e_j), there will be a matching edge, (e_j, e_i)*.*** *Such a digraph is more commonly drawn by combining the pairs of directed edges that connect each pair of vertices by a single undirected edge.* The resulting diagram is called an undirected graph or, more simply a graph. What distinguishes a tree from any other graph is that there are *no circuits*; that is there is no way, beginning at any vertex and by visiting a sequence of two or more distinct vertices (i.e., no repetitions) to return to the same vertex.

Aside: What is an antireflexive relation?

Aside: What is an antireflexive relation?

Let $\mathcal{R} \subseteq S \times S$ be a relation in S.

$\mathcal{R}$ is **antireflexive** iff:

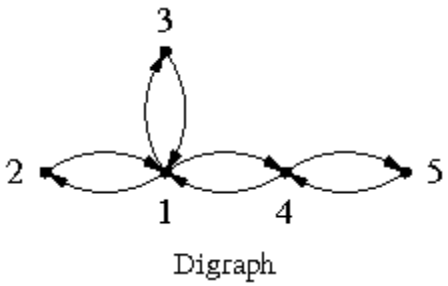
$$\forall x \in S : (x, x) \notin \mathcal{R}$$

Tree Example

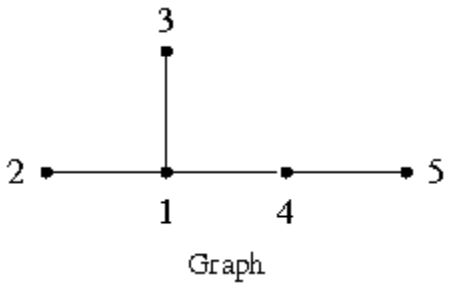
Example:

Given the set A = {1, 2, 3, 4, 5}, consider the binary relation R = {(1, 2), (2, 1), (1, 3), (3, 1), (1, 4), (4, 1), (4, 5), (5, 4)}.

Then the digraph and corresponding graph are:



Digraph



Graph

The graph has no circuits and therefore is a tree.

Equivalent Definitions of a Tree

Let T be a graph with n vertices.

Then the following statetments are equivalent.

- *T is connected and has no cycles.*
- *T has $n-1$ edges and has no cycles.*
- *T is connected and has $n-1$ eges.*
- *T is connected and the removal of any edge disconnects T.*
- *Any two vertices of T are connected by exactly one path.*
- *T contains no cycles, but the additon of any new edge creates a cycle.*


Trees and Forests

Trees and Forests

A **simple path** of length n in a graph is a sequence of distinct vertices v_1, v_2. . .v_n such that v_i is connected by an edge to v_{i+1} for i = 1 to n-1. If a tree is a graph with no circuits, then there is at most one path between any two vertices. Intuitively, if the same two vertices v_a and v_b are connected by two paths, then one can leave v_a via one path to get to v_b and then return to v_a by the other path, completing a circuit of distinct vertices.

Trees and Forests

A **simple path** of length n in a graph is a sequence of distinct vertices v_1, v_2. . .v_n such that v_i is connected by an edge to v_{i+1} for i = 1 to n-1. If a tree is a graph with no circuits, then there is at most one path between any two vertices. Intuitively, if the same two vertices v_a and v_b are connected by two paths, then one can leave v_a via one path to get to v_b and then return to v_a by the other path, completing a circuit of distinct vertices.

Of course, there may be no path between two vertices, v_a and v_b, of a graph. When this is the case then the graph is said to be **disconnected**. In that case all the vertices of the graph can be divided into three sets:

1. Those for which there is a path to v_a;
2. Those for which there is a path to v_b;
3. Those for which there is no path to either v_a or v_b.

Trees and Forests

A **simple path** of length n in a graph is a sequence of distinct vertices v_1, v_2. . .v_n such that v_i is connected by an edge to v_{i+1} for i = 1 to n-1. If a tree is a graph with no circuits, then there is at most one path between any two vertices. Intuitively, if the same two vertices v_a and v_b are connected by two paths, then one can leave v_a via one path to get to v_b and then return to v_a by the other path, completing a circuit of distinct vertices.

Of course, there may be no path between two vertices, v_a and v_b, of a graph. When this is the case then the graph is said to be **disconnected**. In that case all the vertices of the graph can be divided into three sets:

1. Those for which there is a path to v_a;
2. Those for which there is a path to v_b;
3. Those for which there is no path to either v_a or v_b.

These three sets are disjoint and the graphs defined on each subset constitute separate graphs. The graph defined on vertices connected to v_a is **connected**, as is the graph defined on vertices connected to v_b.

Trees and Forests

A **simple path** of length n in a graph is a sequence of distinct vertices v_1, v_2. . .v_n such that v_i is connected by an edge to v_{i+1} for i = 1 to n-1. If a tree is a graph with no circuits, then there is at most one path between any two vertices. Intuitively, if the same two vertices v_a and v_b are connected by two paths, then one can leave v_a via one path to get to v_b and then return to v_a by the other path, completing a circuit of distinct vertices.

Of course, there may be no path between two vertices, v_a and v_b, of a graph. When this is the case then the graph is said to be **disconnected**. In that case all the vertices of the graph can be divided into three sets:

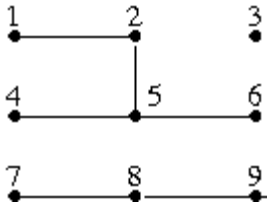
1. Those for which there is a path to v_a;
2. Those for which there is a path to v_b;
3. Those for which there is no path to either v_a or v_b.

These three sets are disjoint and the graphs defined on each subset constitute separate graphs. The graph defined on vertices connected to v_a is **connected**, as is the graph defined on vertices connected to v_b.

The graph defined on vertices not connected to v_a or v_b may or may not be connected. However, one can decompose that graph into connected components in a similar manner. Thus, every graph can be decomposed into a set of one or more connected components. When those components are all trees, then the set of trees is called a **forest**.

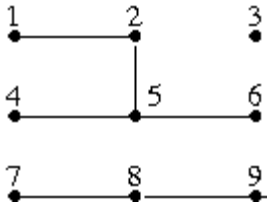
Example

- Consider the following disconnected graph, defined on nine vertices numbered 1 through 9:



Example

- Consider the following disconnected graph, defined on nine vertices numbered 1 through 9:

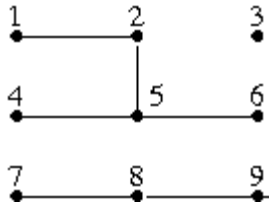


- The vertices can be partitioned into three sets of connected vertices:

{1, 2, 4, 5, 6}
{3}
{7, 8, 9}

Example

- Consider the following disconnected graph, defined on nine vertices numbered 1 through 9:



- The vertices can be partitioned into three sets of connected vertices:

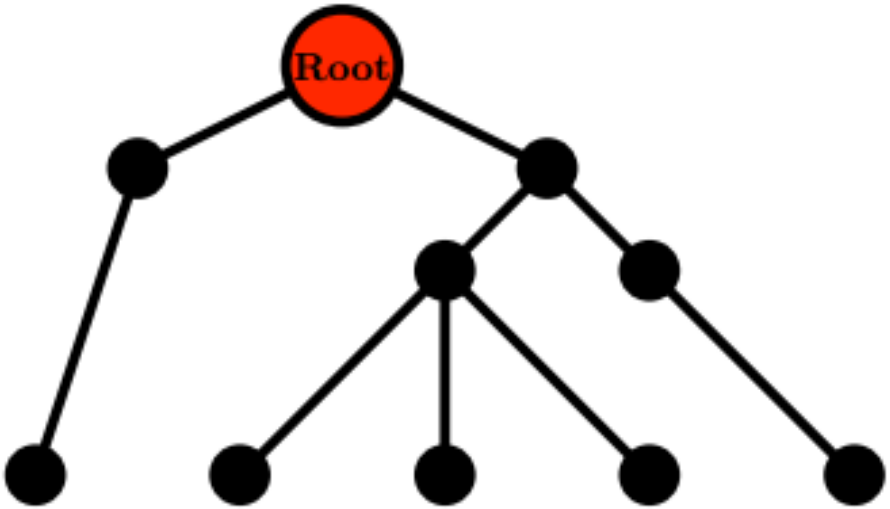
{1, 2, 4, 5, 6}
{3}
{7, 8, 9}

- For each edge in the original, diconnected graph we draw a corresponding edge between the same two vertices in the appropriate component:

a forest of trees

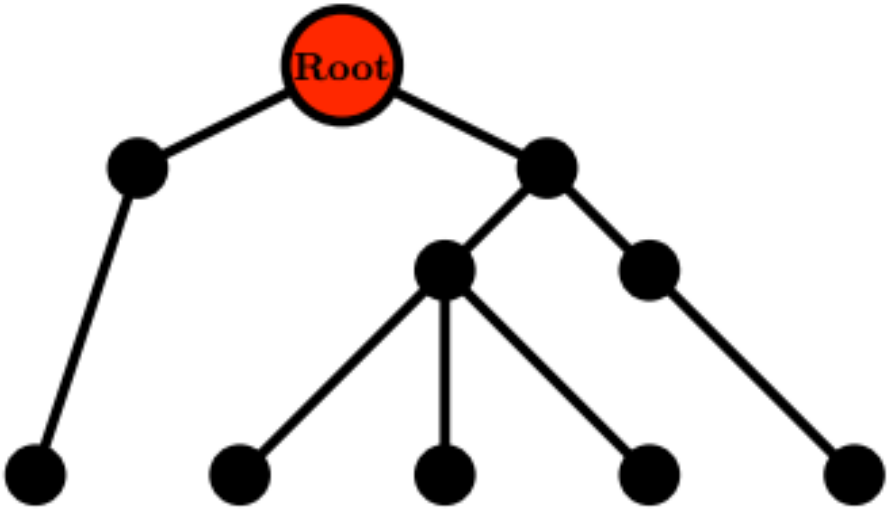
Each component defined on a set of connected vertices is a tree.

Rooted Trees



Rooted Trees

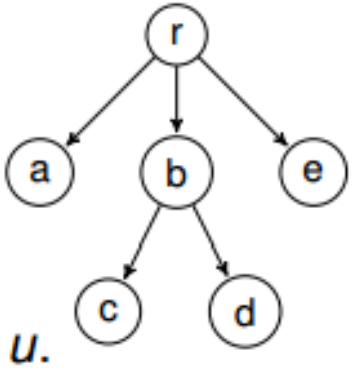
If one vertex of a tree is singled out as a starting point and all the branches fan out from this vertex, we call such a tree a **rooted tree**.



Rooted Tree Summary

Terminology for rooted trees

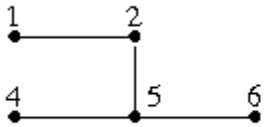
For a rooted tree (T, r), with root r,

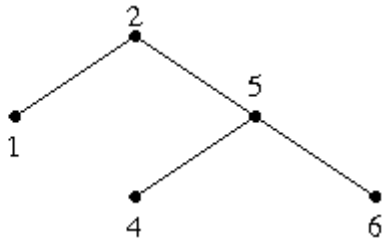


- For each node $v \neq r$ the **parent**, is the unique vertex u such that $(u, v) \in E$. v is then called a **child** of u. Two vertices with the same parent are called **siblings**.
- A **leaf** is a vertex with no children. Non-leaves are called **internal vertices**.
- The **height** of a rooted tree is the length of the longest directed path from the root to any leaf.
- The **ancestors** (**descendants**, respectively) of a vertex v are all vertices $u \neq v$ such that there is a directed path from u to v (from v to u, respectively).
- The **subtree** rooted at v, is the subgraph containing v and all its descendants, and all directed edges between them.

Example

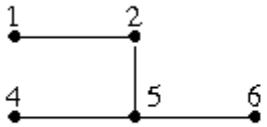
- If the tree given on the left-hand side below has vertex 2 designated as its **root**, then by convention it is often drawn in the manner illustrated on the right-hand side. That is, a **rooted tree** is typically drawn with the root at the top, then all its children are drawn immediately below it, and then all the children of the children are drawn below the children, and so on:

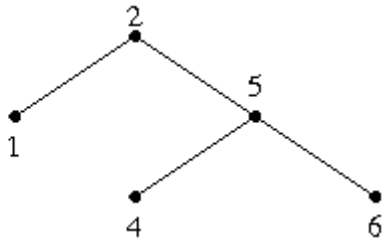




Example

- If the tree given on the left-hand side below has vertex 2 designated as its root, then by convention it is often drawn in the manner illustrated on the right-hand side. That is, a rooted tree is typically drawn with the root at the top, then all its children are drawn immediately below it, and then all the children of the children are drawn below the children, and so on:

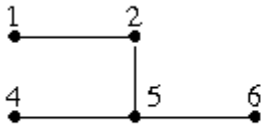


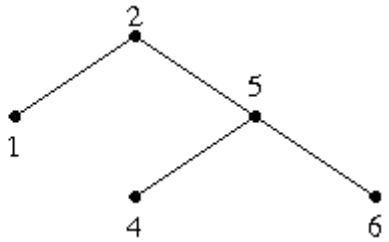


- In this case, 2 is the root, 1 and 5 are its children (making 2 a parent), and 4 and 6 are the children of 5 (making 5 a parent). Vertices 2 and 5 are internal nodes, and vertices 1, 4, and 6 are leaves.

Example

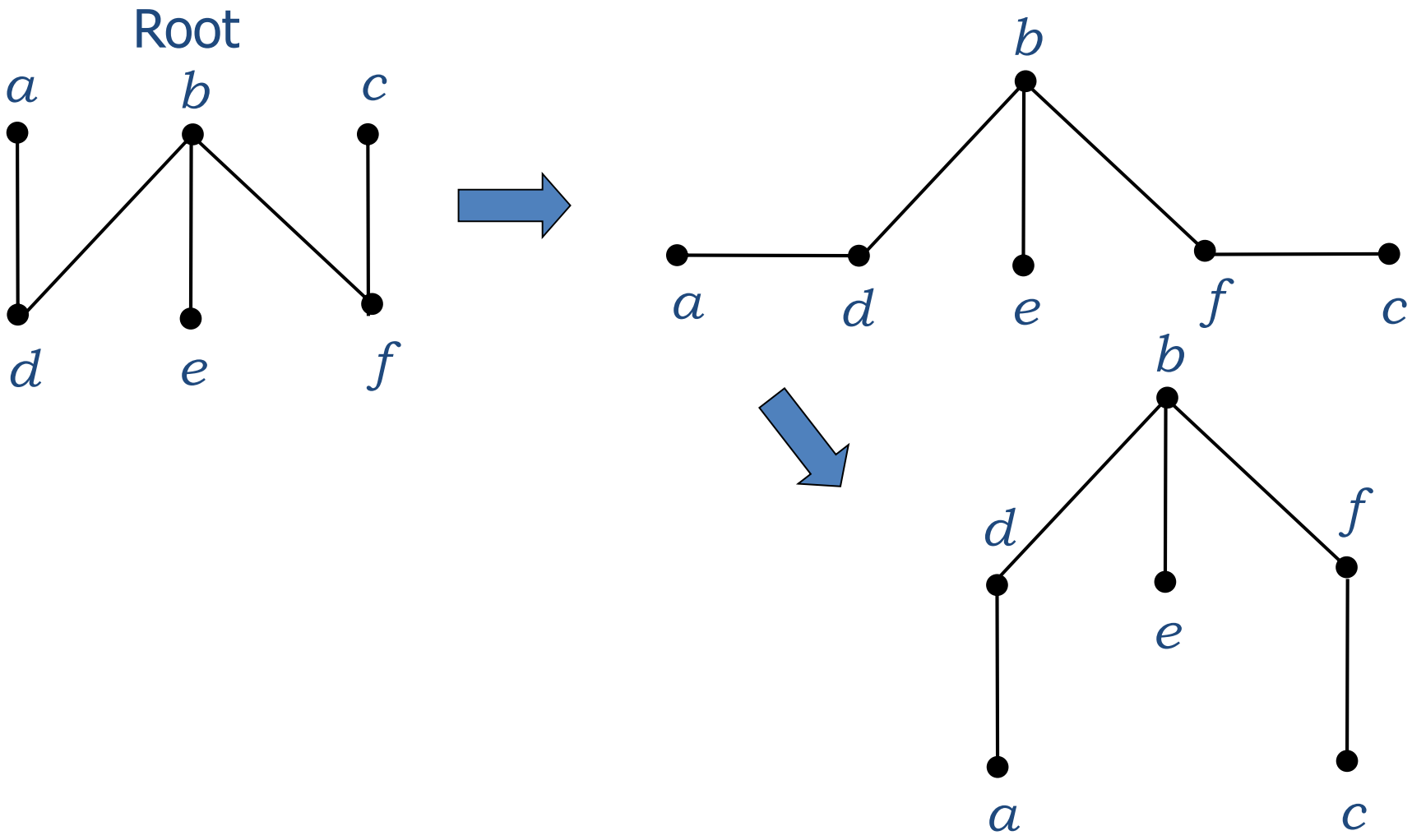
- If the tree given on the left-hand side below has vertex 2 designated as its root, then by convention it is often drawn in the manner illustrated on the right-hand side. That is, a rooted tree is typically drawn with the root at the top, then all its children are drawn immediately below it, and then all the children of the children are drawn below the children, and so on:



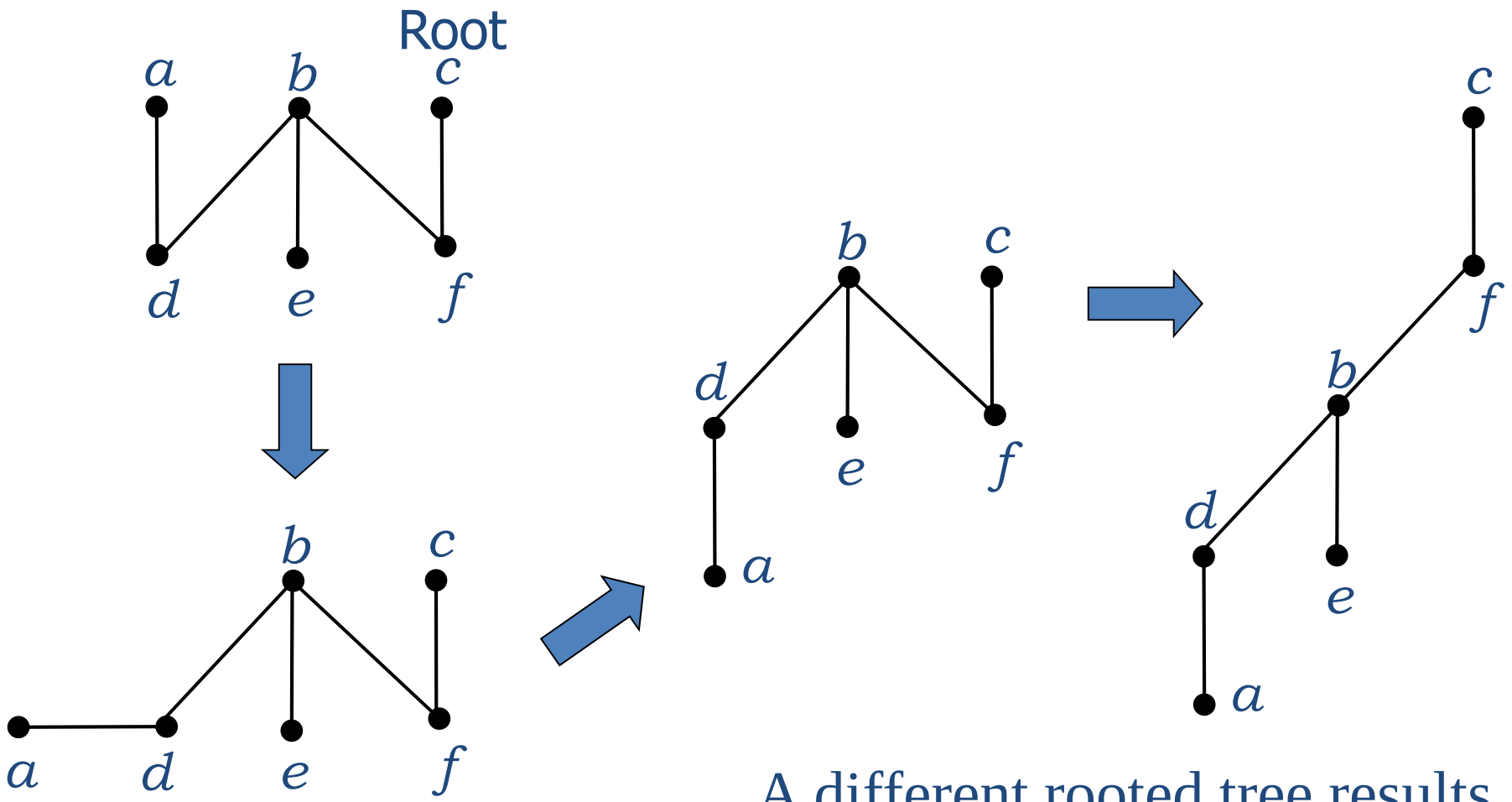


- In this case, 2 is the root, 1 and 5 are its children (making 2 a parent), and 4 and 6 are the children of 5 (making 5 a parent). Vertices 2 and 5 are internal nodes, and vertices 1, 4, and 6 are leaves.
- Except for the leaves, every vertex of a rooted tree has children. If the maximum number of children any vertex has is m, then the rooted tree is said to be an **m-ary rooted tree**. If all internal vertices have m children, the tree is called a **full m-ary rooted tree**. When m = 2 such a tree is called a full binary rooted tree (rather than a full 2-ary tree). The rooted tree with root 2 given in the diagram above is an example of a full binary rooted tree.

Example



What if a different root is chosen?



A different rooted tree results.

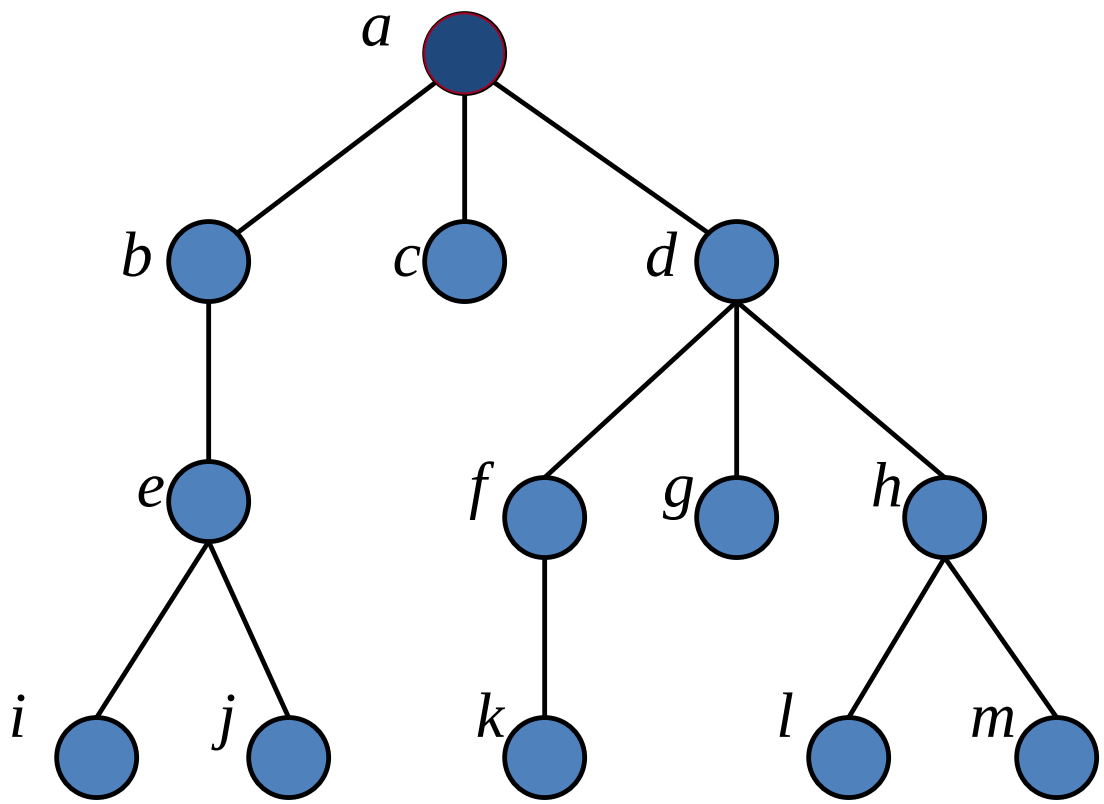
Review of Tree Terminology

Tree Terminology

- If *v* is a vertex of tree *T* 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 the parent of *v*, *v* is called the *child* of *u*.
- If two vertices share the same parent, then they are called *siblings*.

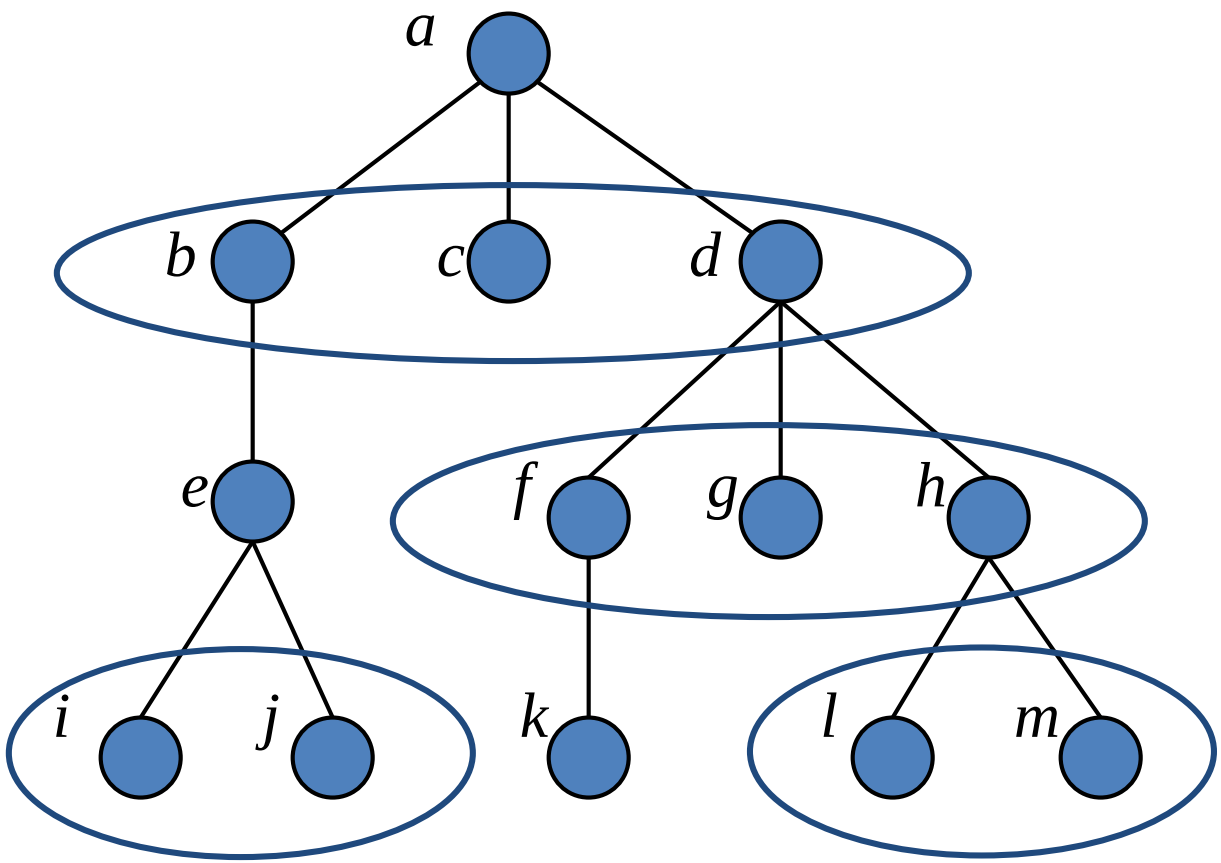
Example

Root



Example

Siblings

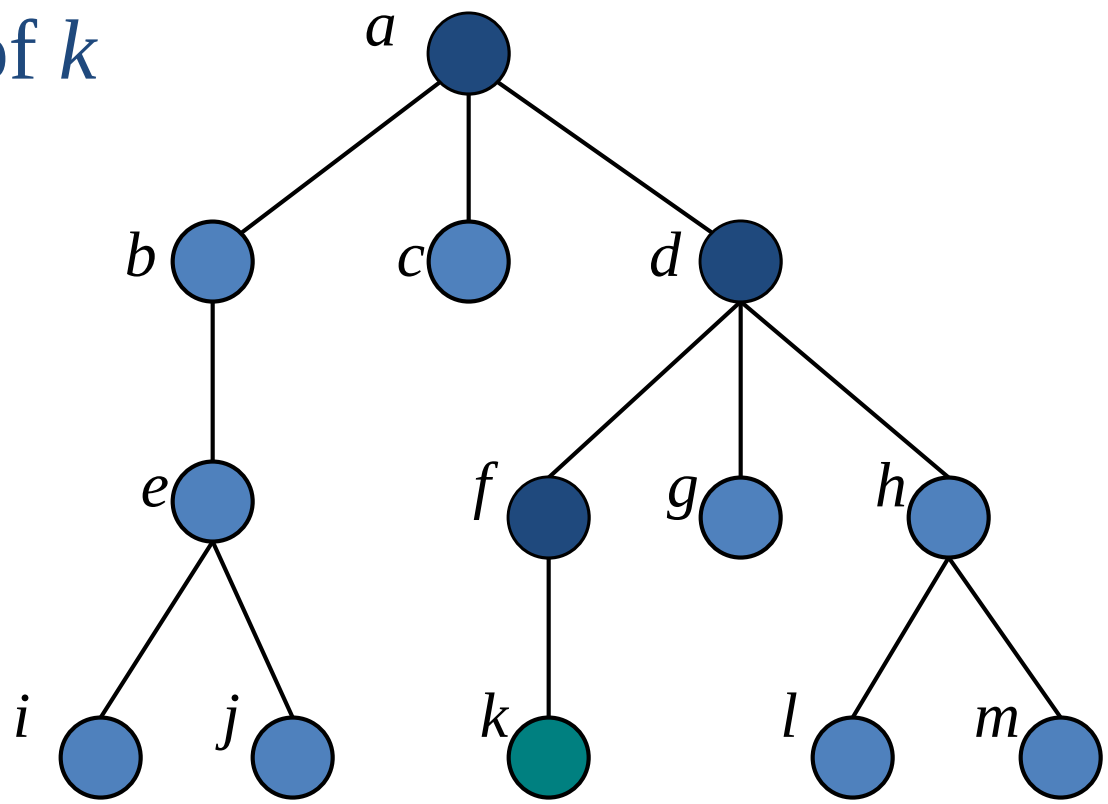


Tree Terminology (Cont.)

- The *ancestors* of a vertex other than the root 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.

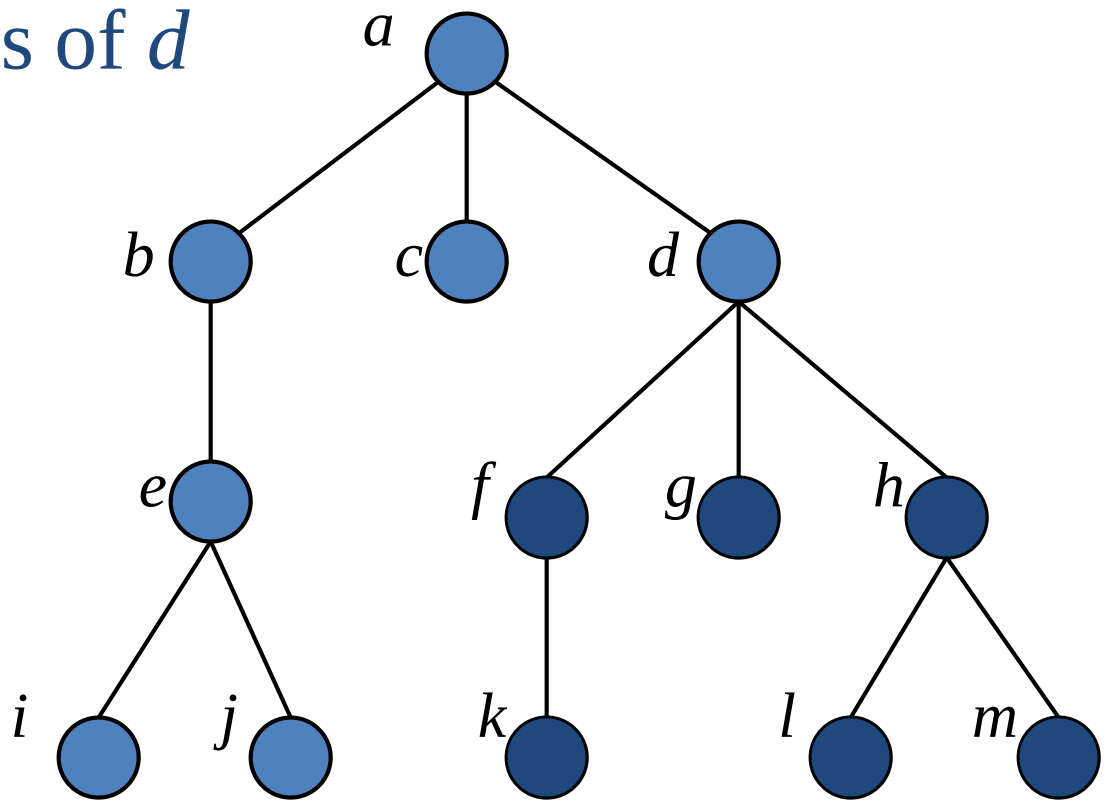
Example

Ancestors of *k*



Example

Descendants of *d*

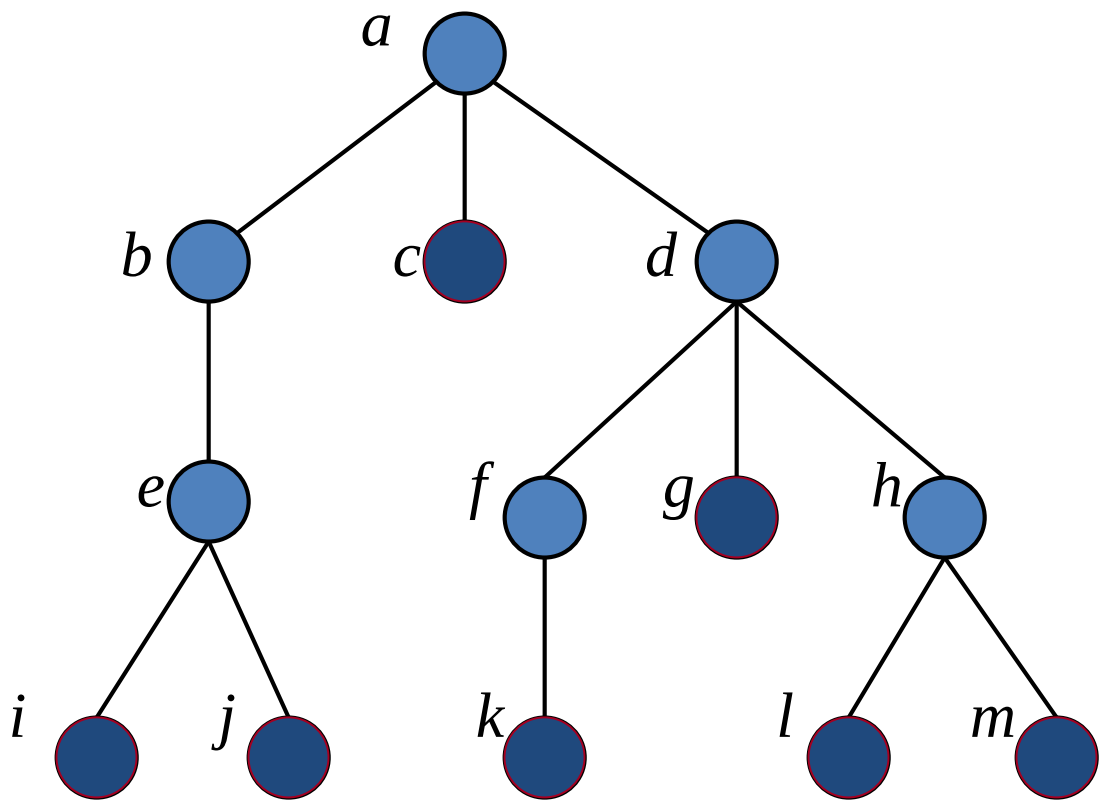


Tree Terminology (Cont.)

- A vertex with no children is called a *leaf*.
- Vertices with children are called *internal vertices*.

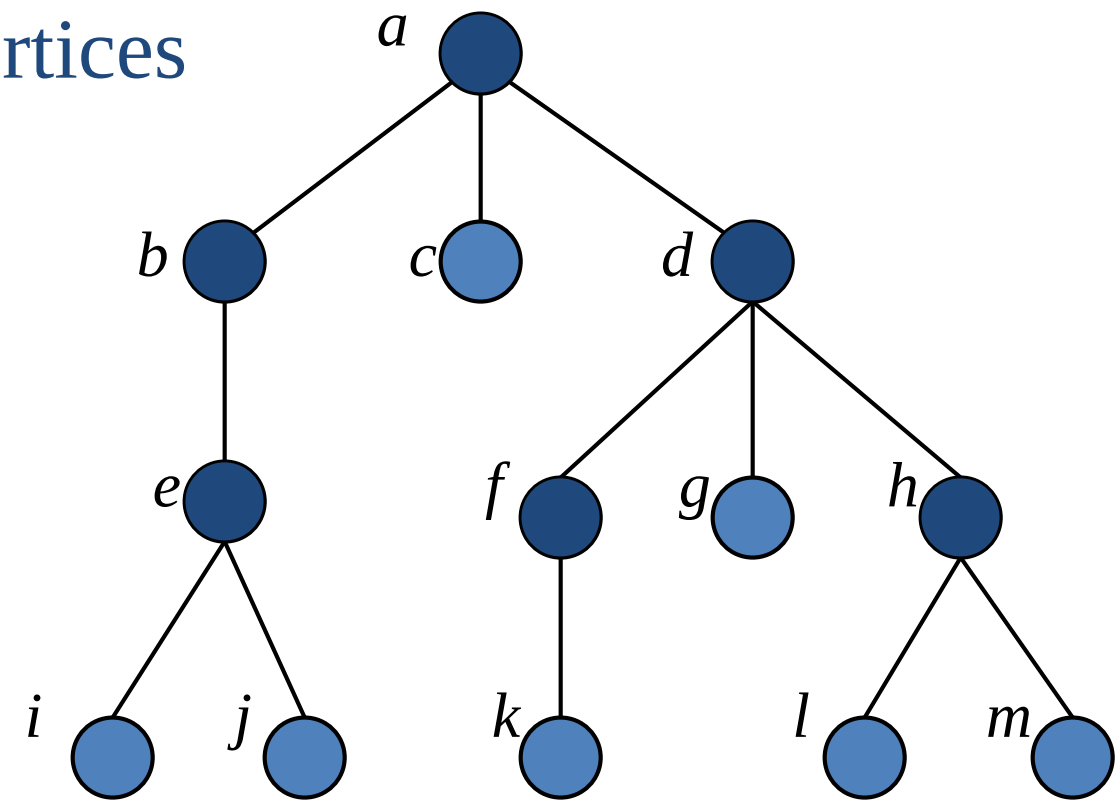
Example

Leaves

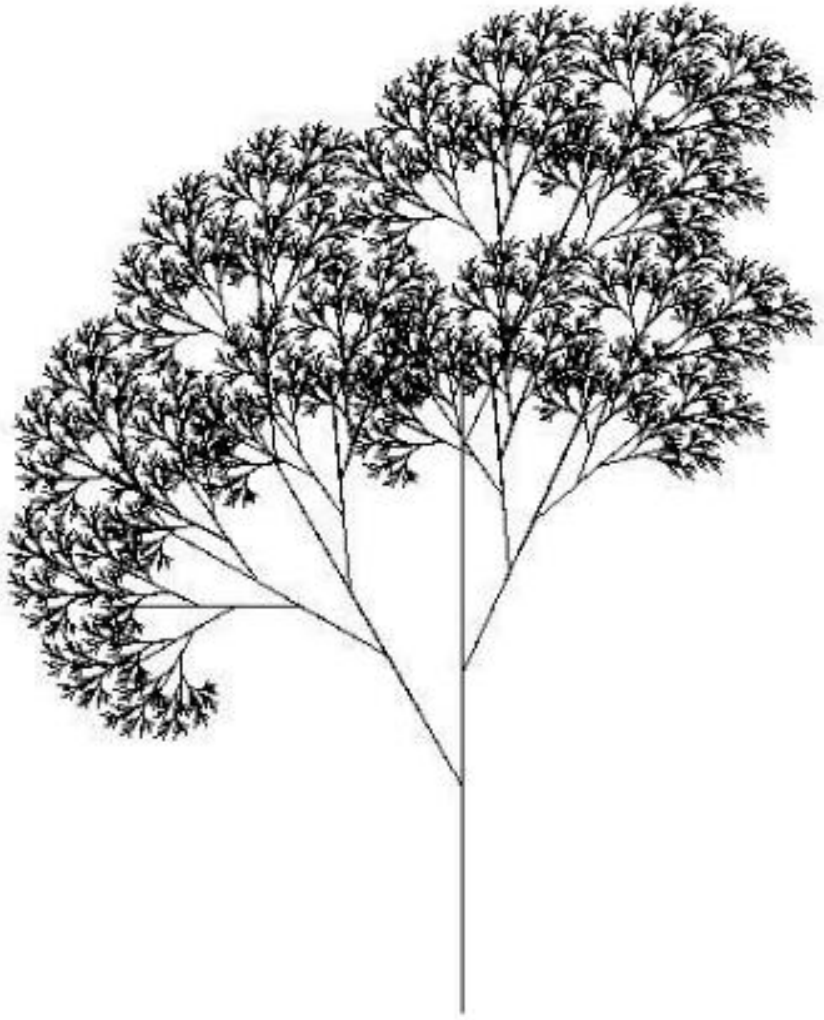


Example

Internal vertices



Recursive Definition of a Rooted Tree



Recursive Definition of a Rooted Tree

The set of *rooted trees,* where a rooted tree consists of a set of vertices containing a distinguished vertex called the *root,* and edges connecting these vertices, can be defined recursively by these steps:

BASIS STEP: A single vertex r is a rooted tree.

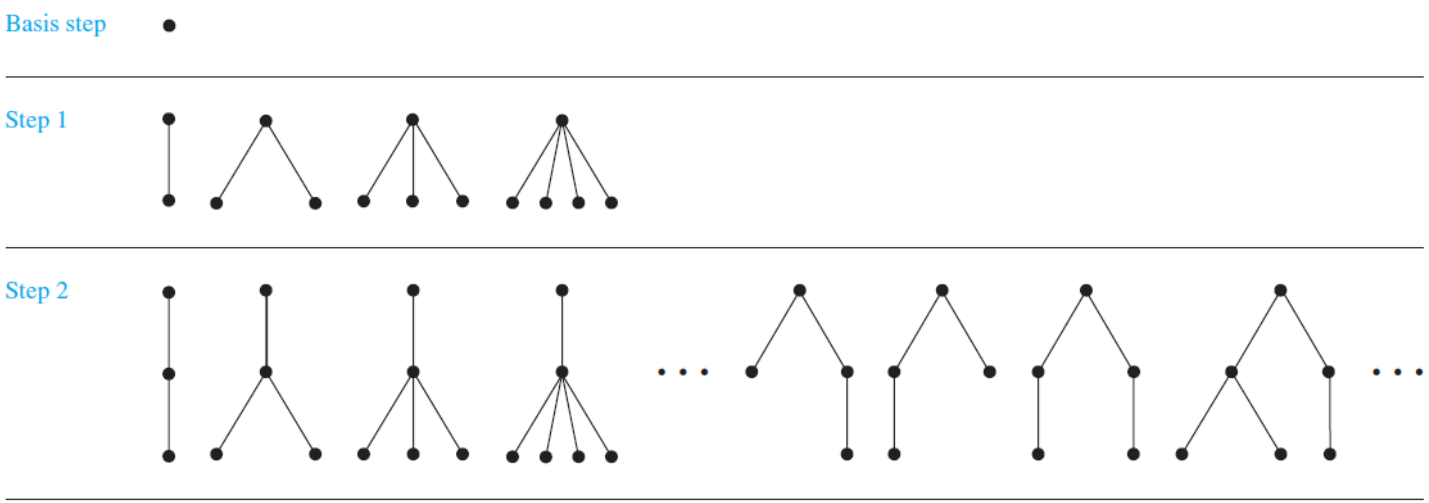
RECURSIVE STEP: Suppose that $T_1, T_2, \ldots, T_n$ are disjoint rooted trees with roots $r_1, r_2, \ldots, r_n$, respectively. Then the graph formed by starting with a root r, which is not in any of the rooted trees $T_1, T_2, \ldots, T_n$, and adding an edge from r to each of the vertices $r_1, r_2, \ldots, r_n$, is also a rooted tree.

Recursive Definition of a Rooted Tree

The set of *rooted trees,* where a rooted tree consists of a set of vertices containing a distinguished vertex called the *root,* and edges connecting these vertices, can be defined recursively by these steps:

BASIS STEP: A single vertex r is a rooted tree.

RECURSIVE STEP: Suppose that $T_1, T_2, \ldots, T_n$ are disjoint rooted trees with roots $r_1, r_2, \ldots, r_n$, respectively. Then the graph formed by starting with a root r, which is not in any of the rooted trees $T_1, T_2, \ldots, T_n$, and adding an edge from r to each of the vertices $r_1, r_2, \ldots, r_n$, is also a rooted tree.



Forests, Subforests, Subtrees and Spanning Trees

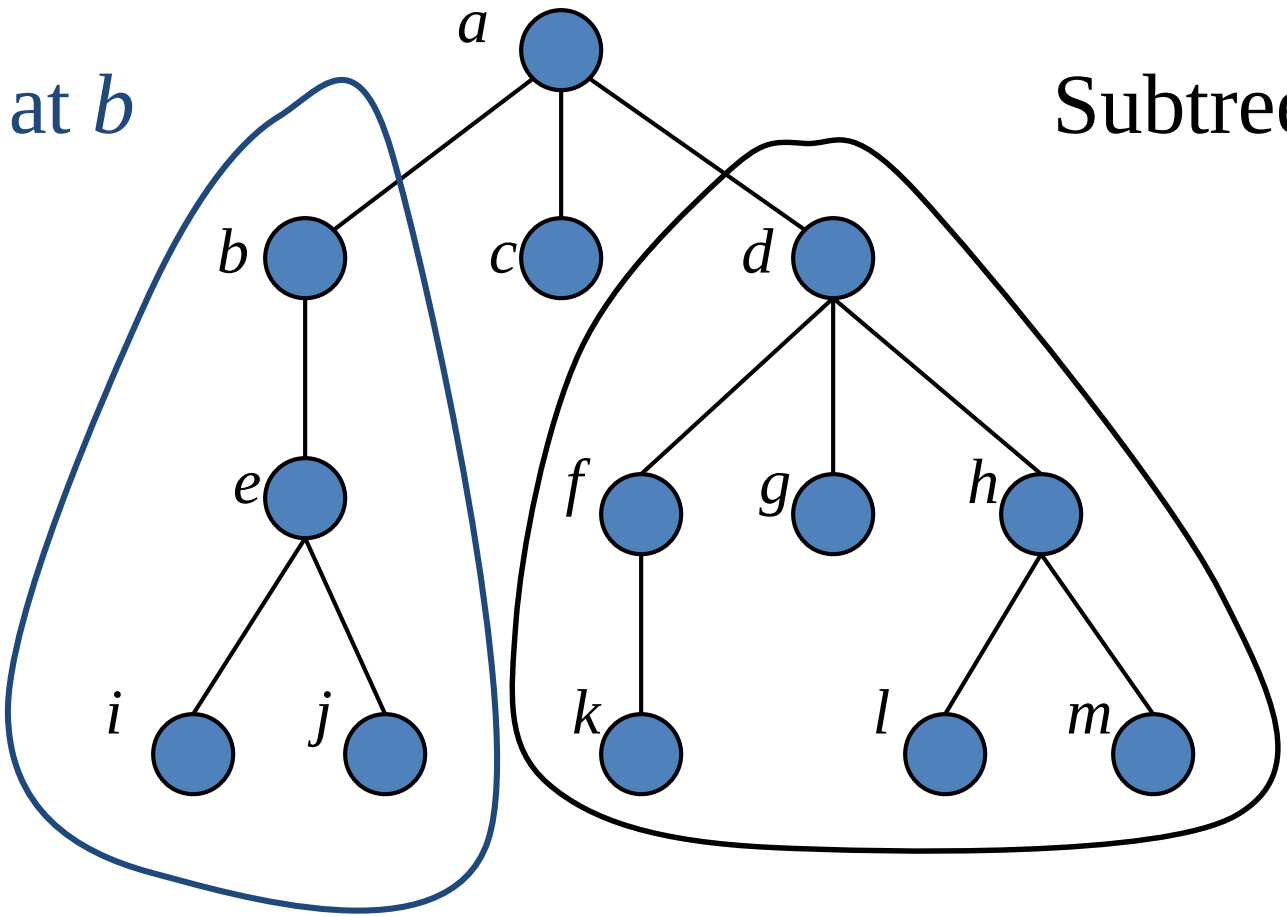
Tree Terminology (Cont.)

- 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.

Example

Subtree at *b*

Subtree at *d*



Forests, Subforests, Subtrees and Spanning Trees

A *forest* is a circuitless graph. A *tree* is a connected forest. A *subforest* is a subgraph of a forest. A connected subgraph of a tree is a *subtree*. Generally speaking, a subforest (respectively subtree) of a graph is its subgraph, which is also a forest (respectively tree).

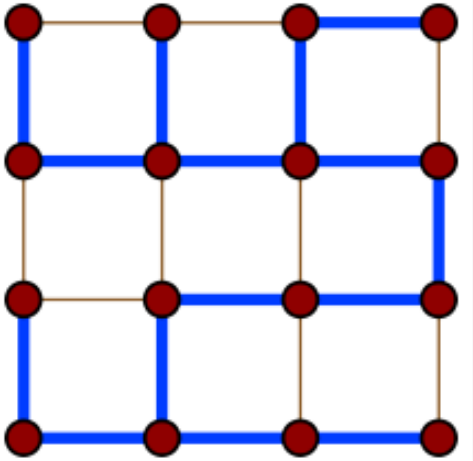
Forests, Subforests, Subtrees and Spanning Trees

A forest is a circuitless graph. *A tree* is a connected forest. *A subforest* is a subgraph of a forest. A connected subgraph of a tree is a *subtree*. Generally speaking, a subforest (respectively subtree) of a graph is its subgraph, which is also a forest (respectively tree).

From Wikipedia, the free encyclopedia

For the protocol used to prevent forwarding loops on a LAN, see Spanning Tree Protocol.

In the mathematical field of graph theory, a **spanning tree** *T* of an undirected graph *G* is a subgraph that is a tree which includes all of the vertices of *G*. In general, a graph may have several spanning trees, but a graph that is not connected will not contain a spanning tree (but see Spanning forests below). If all of the edges of *G* are also edges of a spanning tree *T* of *G*, then *G* is a tree and is identical to *T* (that is, a tree has a unique spanning tree and it is itself).

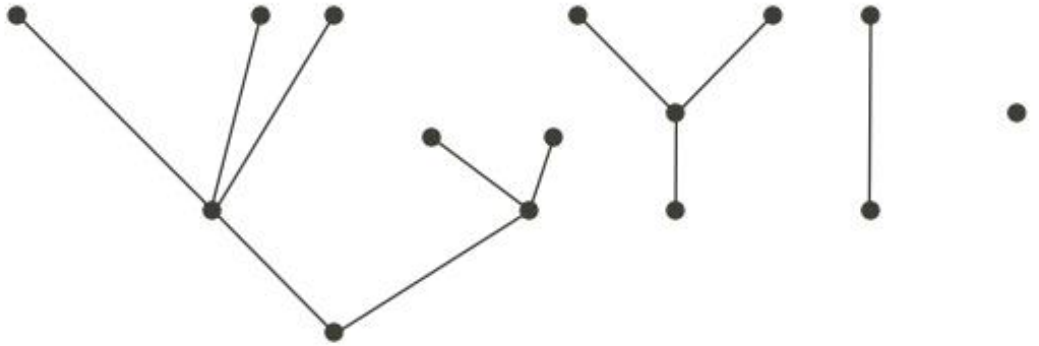


A spanning tree (blue heavy edges) of a grid graph

Forests, Subforests, Subtrees and Spanning Trees

A *forest* is a circuitless graph. A *tree* is a connected forest. A *subforest* is a subgraph of a forest. A connected subgraph of a tree is a *subtree*. Generally speaking, a subforest (respectively subtree) of a graph is its subgraph, which is also a forest (respectively tree).

Example. *Four trees which together form a forest:*



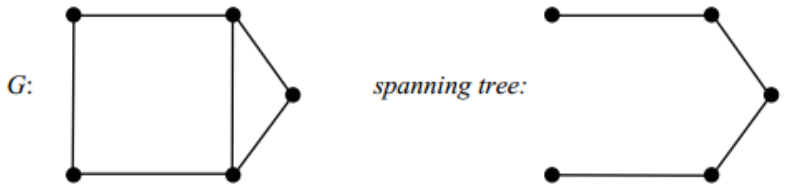
A *spanning tree* of a connected graph is a subtree that includes all the vertices of that graph. If T is a spanning tree of the graph G, then

$$G - T =_{\text{def.}} T^*$$

is the *cospanning tree*.

Spanning Trees

Example.



cospanning tree

The edges of a spanning tree are called *branches* and the edges of the corresponding cospanning tree are called *links* or *chords*.

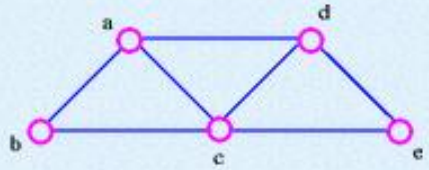
Spanning Trees

We can find a spanning tree systematically by using either of two methods.

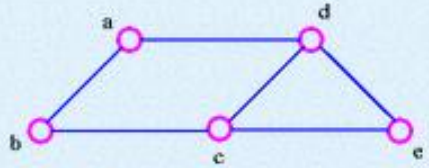
- **Cutting-down Method**
  - Start choosing any cycle in G.
  - Remove one of cycle's edges.
  - Repeat this procedure until there are no cycles left.

Cutting-down Method

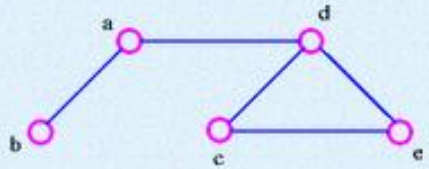
For example, given the graph G



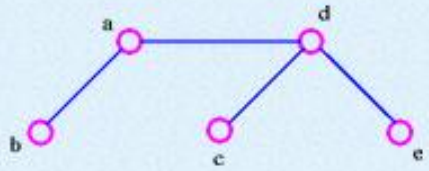
1. We remove the edge ac which destroy the cycle adca in the above graph and we get



2. We remove the edge cb, which destroy the cycle adcba in the above graph and we get



3. We remove the edge ec, which destroy the cycle decd in the above graph and thus obtained the following spanning tree.



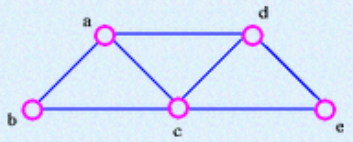
Spanning Trees

We can find a spanning tree systematically by using either of two methods.

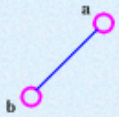
- **Cutting-down Method**
  - Start choosing any cycle in G.
  - Remove one of cycle's edges.
  - Repeat this procedure until there are no cycles left.
- **Building-up Method**
  - Select edges of G one at a time in such a way that no cycles are created.
  - Repeat this procedure until all vertices are included.

Building-up Method

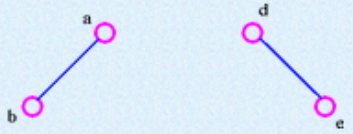
For example, for the following graph G



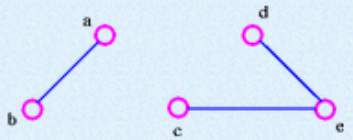
1. Choose the edge ab



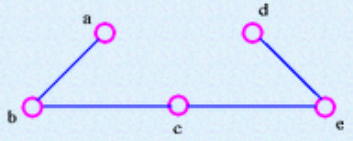
2. Next choose the edge de as follows:



3. After that choose the edge ac as follows:

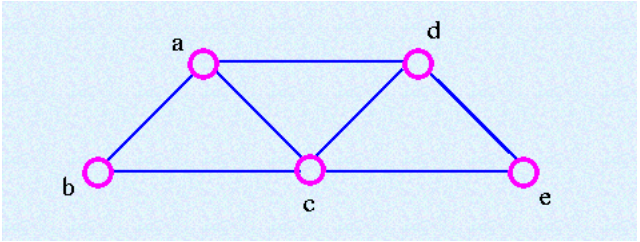


4. Finally, we choose the edge cb and thus obtain the following spanning tree.



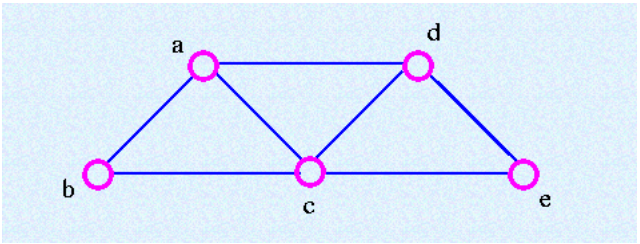
Exercise

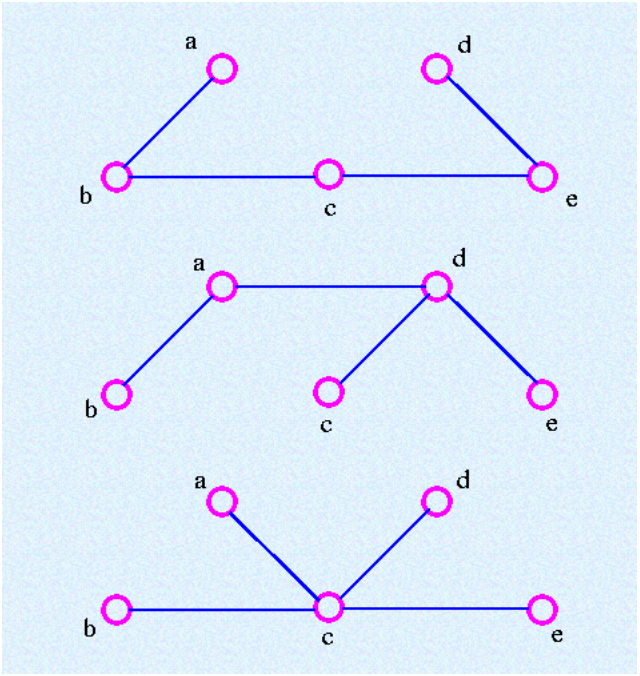
What are the three spanning trees associated with this graph?



Exercise

What are the three spanning trees associated with this graph?





Theorem

A graph is connected if and only if it has a spanning tree.

Theorem

A graph is connected if and only if it has a spanning tree.

Proof

- Let G be a connected graph.
- Delete edges from G that are not bridges until we get a connected subgraph H in which each edge is a bridge.
- Then H is a spanning tree.
- Conversely, if there is a spanning tree in G, there is a path between any pair of vertices in G; thus G is connected.

QED

Counting Edges

- When trees are used in algorithms or to model data structures, it is important to be able to compute various properties possessed by a tree possibly as a function of the number of vertices, number of edges, or number of siblings for example. These parameters and others associated with trees are inter-related; that is, each is often constrained and therefore determined by the values of the others.

Counting Edges

- When trees are used in algorithms or to model data structures, it is important to be able to compute various properties possessed by a tree possibly as a function of the number of vertices, number of edges, or number of siblings for example. These parameters and others associated with trees are inter-related; that is, each is often constrained and therefore determined by the values of the others.
- **In a connected tree, the number of edges, e, is related to the number of vertices, n, by the following theorem:**

$$e = n - 1$$

Counting Edges

- When trees are used in algorithms or to model data structures, it is important to be able to compute various properties possessed by a tree possibly as a function of the number of vertices, number of edges, or number of siblings for example. These parameters and others associated with trees are inter-related; that is, each is often constrained and therefore determined by the values of the others.
- **In a connected tree, the number of edges, e, is related to the number of vertices, n, by the following theorem:**

$$e = n - 1$$

- **We already proved this theorem (Slide #44).**

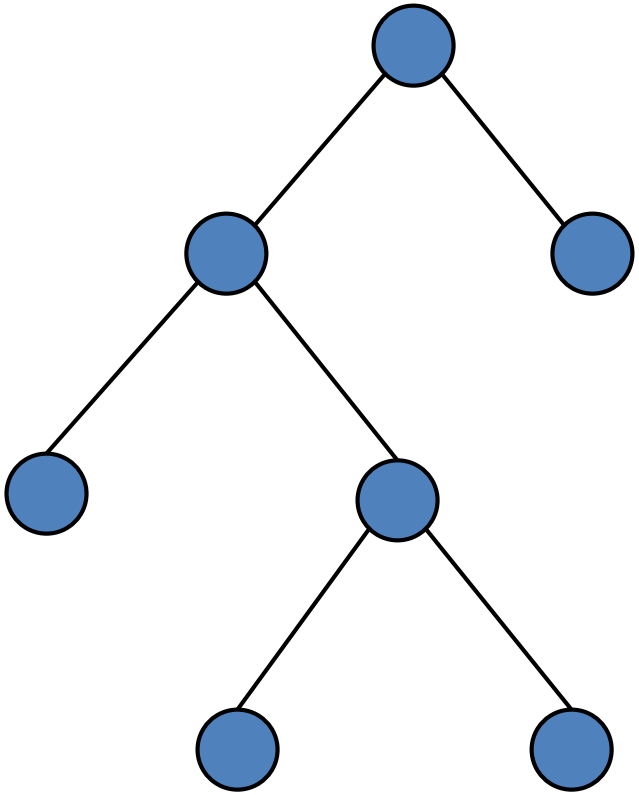
Tree Terminology (Cont.)

- A rooted tree is called an *m-ary tree* if every internal vertex has no more than *m* children.
- A 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

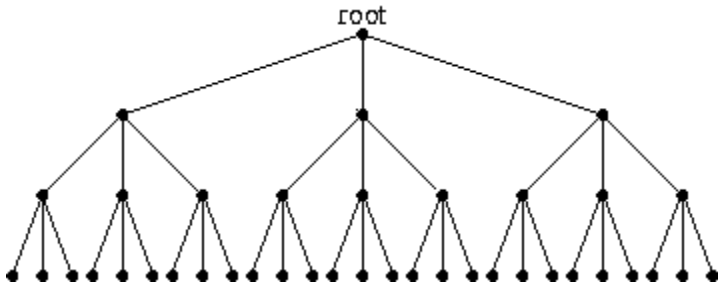
- What is the *arity* of this tree?
- Is this a full *m*-ary tree?

- This is a 2-ary, or *binary*, tree.
- Yes, this is a full binary tree, since every internal vertex has exactly 2 children.



Counting Vertices

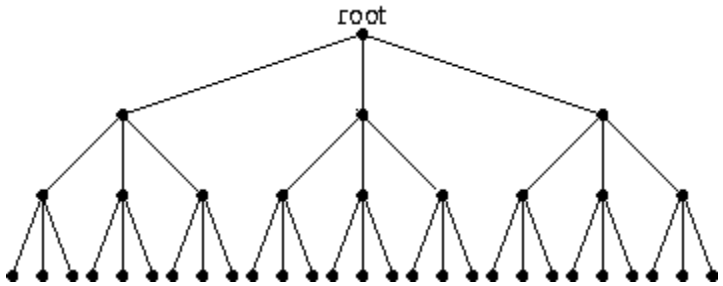
- In rooted trees there are two kinds of vertices: internal vertices and leaves. Many algorithms that employ trees in their implementation use the two types for different purposes. So it is important to recognize the distinction and this includes being able to determine or estimate the number of vertices of each kind that exist in a given tree.



Counting Vertices

- In rooted trees there are two kinds of vertices: internal vertices and leaves. Many algorithms that employ trees in their implementation use the two types for different purposes. So it is important to recognize the distinction and this includes being able to determine or estimate the number of vertices of each kind that exist in a given tree.
- In a full m-ary tree with n vertices, let i be the number of internal vertices. Then the following formula relates these parameters:

$$m \times i = n - 1$$



Counting Vertices

- In rooted trees there are two kinds of vertices: internal vertices and leaves. Many algorithms that employ trees in their implementation use the two types for different purposes. So it is important to recognize the distinction and this includes being able to determine or estimate the number of vertices of each kind that exist in a given tree.
- In a full m-ary tree with n vertices, let i be the number of internal vertices. Then the following formula relates these parameters:

$$m \times i = n - 1$$

- By solving for each parameter in terms of the others we can obtain formulas that count:

Counting Vertices

- In rooted trees there are two kinds of vertices: internal vertices and leaves. Many algorithms that employ trees in their implementation use the two types for different purposes. So it is important to recognize the distinction and this includes being able to determine or estimate the number of vertices of each kind that exist in a given tree.
- In a full m-ary tree with n vertices, let i be the number of internal vertices. Then the following formula relates these parameters:

$$m \times i = n - 1$$

- By solving for each parameter in terms of the others we can obtain formulas that count:
- The number of leaves: $\lambda = n - i = i (m - 1) + 1$.
  NOTE: Because the small letter l is similar to the digit 1, the Greek letter, λ (lambda) will be used to denote the number of leaves in a tree.

Counting Vertices

- In rooted trees there are two kinds of vertices: internal vertices and leaves. Many algorithms that employ trees in their implementation use the two types for different purposes. So it is important to recognize the distinction and this includes being able to determine or estimate the number of vertices of each kind that exist in a given tree.
- In a full m-ary tree with n vertices, let i be the number of internal vertices. Then the following formula relates these parameters:

$$m \times i = n - 1$$

- By solving for each parameter in terms of the others we can obtain formulas that count:
- The number of leaves: $\lambda = n - i = i (m - 1) + 1$.
  NOTE: Because the small letter l is similar to the digit 1, the Greek letter, λ (lambda) will be used to denote the number of leaves in a tree.
- The number of internal vertices: $i = (n - 1) / m$.

Counting Vertices

- In rooted trees there are two kinds of vertices: internal vertices and leaves. Many algorithms that employ trees in their implementation use the two types for different purposes. So it is important to recognize the distinction and this includes being able to determine or estimate the number of vertices of each kind that exist in a given tree.
- In a full m-ary tree with n vertices, let i be the number of internal vertices. Then the following formula relates these parameters:

$$m \times i = n - 1$$

- By solving for each parameter in terms of the others we can obtain formulas that count:
- The number of leaves: $\lambda = n - i = i (m - 1) + 1$.
  NOTE: Because the small letter l is similar to the digit 1, the Greek letter, λ (lambda) will be used to denote the number of leaves in a tree.
- The number of internal vertices: $i = (n - 1) / m$.
- The total number of vertices: $n = m \times i + 1$.

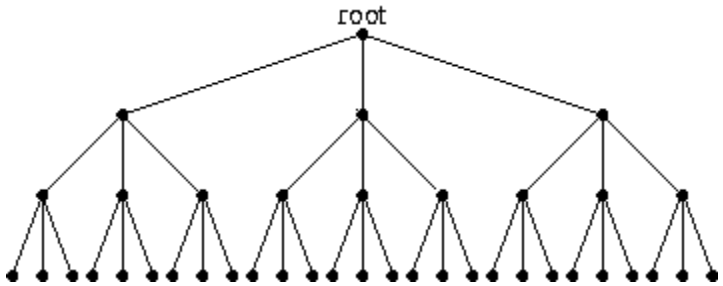
Counting Vertices

- In rooted trees there are two kinds of vertices: internal vertices and leaves. Many algorithms that employ trees in their implementation use the two types for different purposes. So it is important to recognize the distinction and this includes being able to determine or estimate the number of vertices of each kind that exist in a given tree.
- In a full m-ary tree with n vertices, let i be the number of internal vertices. Then the following formula relates these parameters:

$$m \times i = n - 1$$

- By solving for each parameter in terms of the others we can obtain formulas that count:
- The number of leaves: $\lambda = n - i = i (m - 1) + 1$.
  NOTE: Because the small letter l is similar to the digit 1, the Greek letter, λ (lambda) will be used to denote the number of leaves in a tree.
- The number of internal vertices: $i = (n - 1) / m$.
- The total number of vertices: $n = m \times i + 1$.
- By combining formula (3) with the edge counting formula, we can obtain a formula for the number of edges, e, in a full m-ary rooted tree with i internal vertices:

$$e = m \times i$$



Counting Vertices

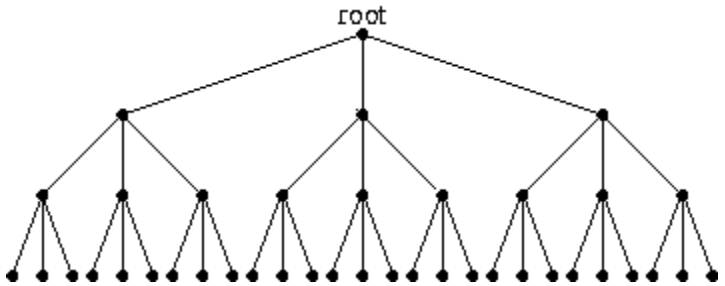
- In rooted trees there are two kinds of vertices: internal vertices and leaves. Many algorithms that employ trees in their implementation use the two types for different purposes. So it is important to recognize the distinction and this includes being able to determine or estimate the number of vertices of each kind that exist in a given tree.
- In a full m-ary tree with n vertices, let i be the number of internal vertices. Then the following formula relates these parameters:

$$m \times i = n - 1$$

- By solving for each parameter in terms of the others we can obtain formulas that count:
- The number of leaves: $\lambda = n - i = i (m - 1) + 1$.
  NOTE: Because the small letter l is similar to the digit 1, the Greek letter, λ (lambda) will be used to denote the number of leaves in a tree.
- The number of internal vertices: $i = (n - 1) / m$.
- The total number of vertices: $n = m \times i + 1$.
- By combining formula (3) with the edge counting formula, we can obtain a formula for the number of edges, e, in a full m-ary rooted tree with i internal vertices:

$$e = m \times i$$

- The following tree is an example of a full ternary (i.e. 3-ary) rooted tree:



- By inspection, there are:
- e = 39 edges;
- n = 40 vertices;
- m = 3 children for every internal vertex;
- i = 13 internal vertices;
- λ = 27 leaves
- Then, $m \times i = 3 \times 13 = 39 = n - 1$

Balanced Trees

- A common activity in algorithms that employ trees to represent data structures is traversal of the tree from its root to some designated vertex, either an internal one or a leaf. In a rooted tree there is a unique path from the root to any vertex. The **length** of any path from v_a to v_b is the number of edges in the path connecting v_a to v_b.

Balanced Trees

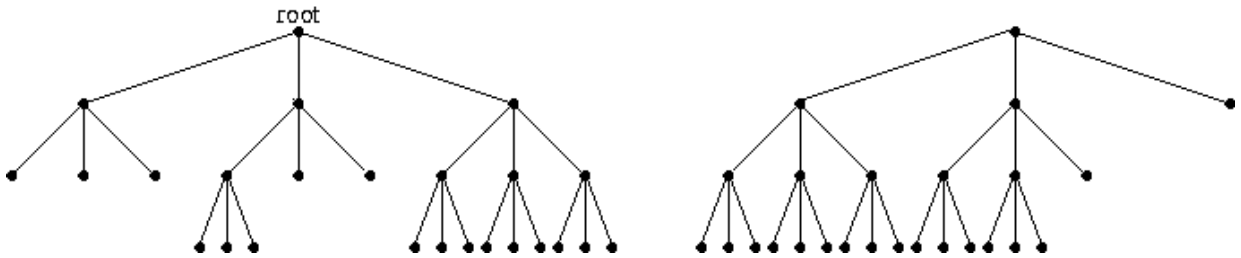
- A common activity in algorithms that employ trees to represent data structures is traversal of the tree from its root to some designated vertex, either an internal one or a leaf. In a rooted tree there is a unique path from the root to any vertex. The **length** of any path from v_a to v_b is the number of edges in the path connecting v_a to v_b.
- The **level** of a vertex in a rooted tree is its distance from the root vertex. Hence the level of the root vertex is 0, since there are no edges connecting the root to itself (Remember, there are no loops in the graph of a tree.)

Balanced Trees

- A common activity in algorithms that employ trees to represent data structures is traversal of the tree from its root to some designated vertex, either an internal one or a leaf. In a rooted tree there is a unique path from the root to any vertex. The **length** of any path from v_a to v_b is the number of edges in the path connecting v_a to v_b.
- The **level** of a vertex in a rooted tree is its distance from the root vertex. Hence the level of the root vertex is 0, since there are no edges connecting the root to itself (Remember, there are no loops in the graph of a tree.)
- The **height** of a tree is the length of the longest path from the root to any leaf. Note that this is equivalent to the maximum level of any leaf.

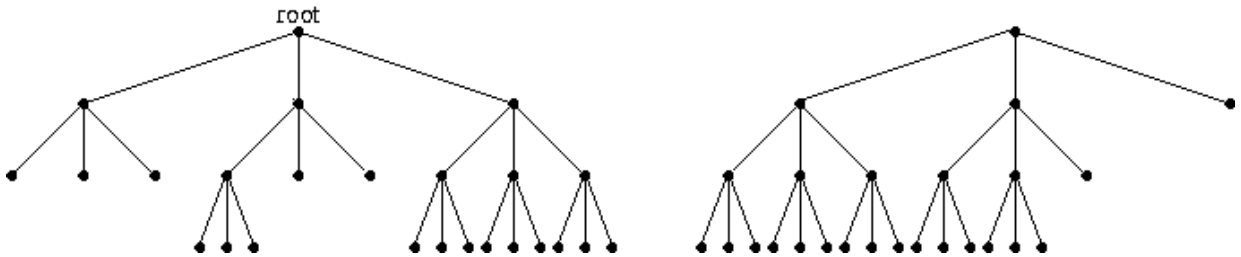
Balanced Trees

- A common activity in algorithms that employ trees to represent data structures is traversal of the tree from its root to some designated vertex, either an internal one or a leaf. In a rooted tree there is a unique path from the root to any vertex. The **length** of any path from v_a to v_b is the number of edges in the path connecting v_a to v_b.
- The **level** of a vertex in a rooted tree is its distance from the root vertex. Hence the level of the root vertex is 0, since there are no edges connecting the root to itself (Remember, there are no loops in the graph of a tree.)
- The **height** if a tree is the length of the longest path from the root to any leaf. Note that this is equivalent to the maximum level of any leaf.
- The following trees represent two full ternary rooted trees:



Balanced Trees

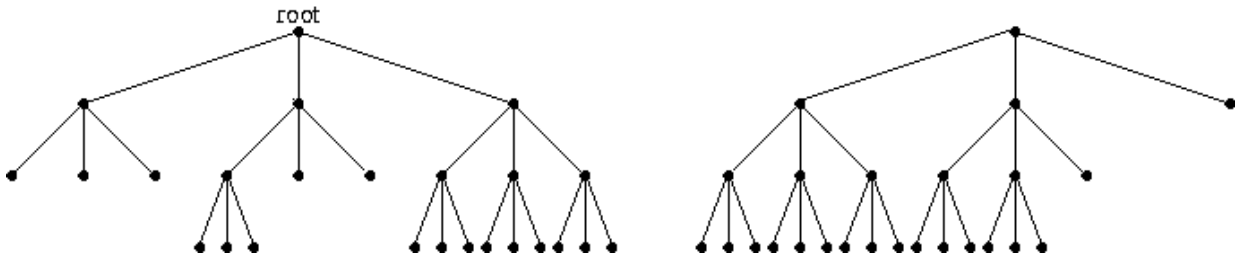
- A common activity in algorithms that employ trees to represent data structures is traversal of the tree from its root to some designated vertex, either an internal one or a leaf. In a rooted tree there is a unique path from the root to any vertex. The **length** of any path from v_a to v_b is the number of edges in the path connecting v_a to v_b.
- The **level** of a vertex in a rooted tree is its distance from the root vertex. Hence the level of the root vertex is 0, since there are no edges connecting the root to itself (Remember, there are no loops in the graph of a tree.)
- The **height** if a tree is the length of the longest path from the root to any leaf. Note that this is equivalent to the maximum level of any leaf.
- The following trees represent two full ternary rooted trees:



- By inspection, for both trees there are:
  e = 24 edges;
  n = 25 vertices;
  m = 3 children for every internal vertex;
  i = 8 internal vertices;
  λ = 17 leaves

Balanced Trees

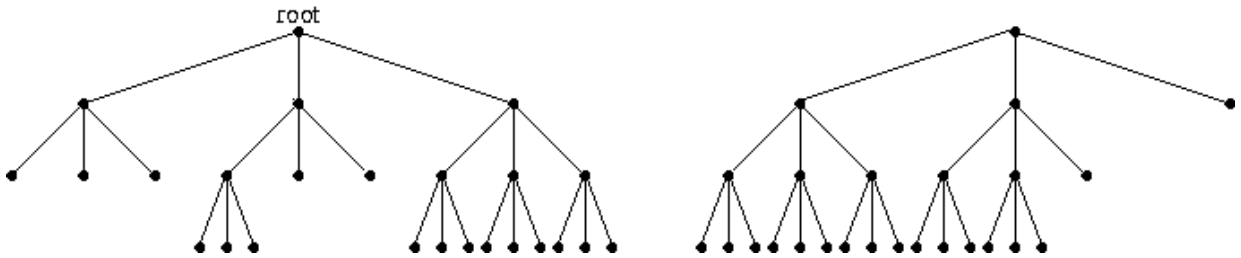
- A common activity in algorithms that employ trees to represent data structures is traversal of the tree from its root to some designated vertex, either an internal one or a leaf. In a rooted tree there is a unique path from the root to any vertex. The **length** of any path from v_a to v_b is the number of edges in the path connecting v_a to v_b.
- The **level** of a vertex in a rooted tree is its distance from the root vertex. Hence the level of the root vertex is 0, since there are no edges connecting the root to itself (Remember, there are no loops in the graph of a tree.)
- The **height** if a tree is the length of the longest path from the root to any leaf. Note that this is equivalent to the maximum level of any leaf.
- The following trees represent two full ternary rooted trees:



- By inspection, for both trees there are:
  - e = 24 edges;
  - n = 25 vertices;
  - m = 3 children for every internal vertex;
  - i = 8 internal vertices;
  - λ = 17 leaves
- Each tree has height = 3. However, the left tree has five level-2 leaves whereas the right tree has one level-2 leaf and one level-1 leaf.

Balanced Trees

- A common activity in algorithms that employ trees to represent data structures is traversal of the tree from its root to some designated vertex, either an internal one or a leaf. In a rooted tree there is a unique path from the root to any vertex. The **length** of any path from v_a to v_b is the number of edges in the path connecting v_a to v_b.
- The **level** of a vertex in a rooted tree is its distance from the root vertex. Hence the level of the root vertex is 0, since there are no edges connecting the root to itself (Remember, there are no loops in the graph of a tree.)
- The **height** if a tree is the length of the longest path from the root to any leaf. Note that this is equivalent to the maximum level of any leaf.
- The following trees represent two full ternary rooted trees:



- By inspection, for both trees there are:
  - e = 24 edges;
  - n = 25 vertices;
  - m = 3 children for every internal vertex;
  - i = 8 internal vertices;
  - λ = 17 leaves
- Each tree has height = 3. However, the left tree has five level-2 leaves whereas the right tree has one level-2 leaf and one level-1 leaf.
- In some algorithms that employ trees, it is important that all paths from the root to the leaves be nearly the same length. In particular, a tree of height h where the level of all leaves is either h or h - 1 is called a **balanced** tree. Therefore the left tree above is a balanced tree, whereas the right tree is not.

Binary Trees

- Deciding between two alternatives is a very common activity. This is because logical reasoning is the basis of formal decision making. Given a declarative statement, we must decide whether it is true or false. When a sequence of such decisions must be made, the choices and outcomes can be represented by a particular type of rooted tree, called a binary tree.

Binary Trees

- Deciding between two alternatives is a very common activity. This is because logical reasoning is the basis of formal decision making. Given a declarative statement, we must decide whether it is true or false. When a sequence of such decisions must be made, the choices and outcomes can be represented by a particular type of rooted tree, called a binary tree.
- As a consequence, the properties and analysis of binary trees are of particular interest. Among those properties are the number of leaves and the height of a binary tree. These two properties are related in the following way: In a binary tree of height h and having λ leaves:

$$\lambda \leq 2^h$$

Binary Trees

- Deciding between two alternatives is a very common activity. This is because logical reasoning is the basis of formal decision making. Given a declarative statement, we must decide whether it is true or false. When a sequence of such decisions must be made, the choices and outcomes can be represented by a particular type of rooted tree, called a binary tree.
- As a consequence, the properties and analysis of binary trees are of particular interest. Among those properties are the number of leaves and the height of a binary tree. These two properties are related in the following way: In a binary tree of height h and having λ leaves:

$$\lambda \leq 2^h$$

- One way to verify this result is to construct a recursive definition for the number of leaves in a complete binary tree of height h. Observe that to obtain a complete binary tree of height h from a complete binary tree of height h - 1, it is necessary to add two new leaves to each leaf of the complete binary tree of height h - 1.

Binary Trees

- Deciding between two alternatives is a very common activity. This is because logical reasoning is the basis of formal decision making. Given a declarative statement, we must decide whether it is true or false. When a sequence of such decisions must be made, the choices and outcomes can be represented by a particular type of rooted tree, called a binary tree.
- As a consequence, the properties and analysis of binary trees are of particular interest. Among those properties are the number of leaves and the height of a binary tree. These two properties are related in the following way: In a binary tree of height h and having λ leaves:

$$\lambda \leq 2^h$$

- One way to verify this result is to construct a recursive definition for the number of leaves in a complete binary tree of height h. Observe that to obtain a complete binary tree of height h from a complete binary tree of height h - 1, it is necessary to add two new leaves to each leaf of the complete binary tree of height h - 1.
- Denote by λ_h the number of leaves in a complete binary tree of height h.

Binary Trees

- Deciding between two alternatives is a very common activity. This is because logical reasoning is the basis of formal decision making. Given a declarative statement, we must decide whether it is true or false. When a sequence of such decisions must be made, the choices and outcomes can be represented by a particular type of rooted tree, called a binary tree.
- As a consequence, the properties and analysis of binary trees are of particular interest. Among those properties are the number of leaves and the height of a binary tree. These two properties are related in the following way: In a binary tree of height h and having λ leaves:

$$\lambda \leq 2^h$$

- One way to verify this result is to construct a recursive definition for the number of leaves in a complete binary tree of height h. Observe that to obtain a complete binary tree of height h from a complete binary tree of height h - 1, it is necessary to add two new leaves to each leaf of the complete binary tree of height h - 1.
- Denote by λ_h the number of leaves in a complete binary tree of height h.
- Base Rule: $\lambda_0 = 1$
- Recursive Rule: For $h > 0$, $\lambda_h = 2 \times \lambda_{h-1}$

Binary Trees

- The number of leaves in the complete binary trees of heights 1, 2, and 3 are: $\lambda_1 = 2^1 = 2$, $\lambda_2 = 2^2 = 4$, and $\lambda_3 = 2^3 = 8$. Using informal inductive reasoning, we can hypothesize that $\lambda_h = 2^h$. Once again, we can apply mathematical induction to formally prove this hypothesis:

Binary Trees

- The number of leaves in the complete binary trees of heights 1, 2, and 3 are: $\lambda_1 = 2^1 = 2$, $\lambda_2 = 2^2 = 4$, and $\lambda_3 = 2^e = 8$. Using informal inductive reasoning, we can hypothesize that $\lambda_h = 2^h$. Once again, we can apply mathematical induction to formally prove this hypothesis:
- **Basis Step:** To prove $\lambda_0 = 2^0$
  - Proof: 1. $\lambda_0 = 1 = 2^0$

Binary Trees

- The number of leaves in the complete binary trees of heights 1, 2, and 3 are: $\lambda_1 = 2^1 = 2$, $\lambda_2 = 2^2 = 4$, and $\lambda_3 = 2^e = 8$. Using informal inductive reasoning, we can hypothesize that $\lambda_h = 2^h$. Once again, we can apply mathematical induction to formally prove this hypothesis:
- **Basis Step:** To prove $\lambda_0 = 2^0$
  - Proof: 1. $\lambda_0 = 1 = 2^0$
- **Inductive Hypothesis:** Assume $\lambda_h = 2^h$

Binary Trees

- The number of leaves in the complete binary trees of heights 1, 2, and 3 are: $\lambda_1 = 2^1 = 2$, $\lambda_2 = 2^2 = 4$, and $\lambda_3 = 2^e = 8$. Using informal inductive reasoning, we can hypothesize that $\lambda_h = 2^h$. Once again, we can apply mathematical induction to formally prove this hypothesis:
- **Basis Step:** To prove $\lambda_0 = 2^0$
  - Proof: 1. $\lambda_0 = 1 = 2^0$
- **Inductive Hypothesis:** Assume $\lambda_h = 2^h$
- **Inductive Step**: To prove $\lambda_{h+1} = 2^{h+1}$
  - Proof: 1. $\lambda_{h+1} = 2 \times \lambda_h$ from the recursive definition 2. $= 2 \times 2^h$ by substitution using the inductive hypothesis $= 2^{h+1}$
- Therefore the number of leaves in a complete binary tree of height h is given by:

$\lambda = 2^h$ **QED**

Binary Trees

- The number of leaves in the complete binary trees of heights 1, 2, and 3 are: $\lambda_1 = 2^1 = 2$, $\lambda_2 = 2^2 = 4$, and $\lambda_3 = 2^e = 8$. Using informal inductive reasoning, we can hypothesize that $\lambda_h = 2^h$. Once again, we can apply mathematical induction to formally prove this hypothesis:
- **Basis Step:** To prove $\lambda_0 = 2^0$
  - Proof: 1. $\lambda_0 = 1 = 2^0$
- **Inductive Hypothesis:** Assume $\lambda_h = 2^h$
- **Inductive Step**: To prove $\lambda_{h+1} = 2^{h+1}$
  - Proof: 1. $\lambda_{h+1} = 2 \times \lambda_h$ from the recursive definition 2. $= 2 \times 2^h$ by substitution using the inductive hypothesis 3. $= 2^{h+1}$
- Therefore the number of leaves in a complete binary tree of height h is given by:

$$\lambda = 2^h \quad \textbf{QED}$$

- The importance of this result is that it places a cap on the number of leaves that can occur in a binary tree, with that cap determined by the height of the tree. In effect, for a binary tree of height h, there can be at most 2^h leaves. To illustrate the significance, consider the problem of growing a binary tree; that is, adding new vertices to it. There are two ways to do this:

Binary Trees

- The number of leaves in the complete binary trees of heights 1, 2, and 3 are: $\lambda_1 = 2^1 = 2$, $\lambda_2 = 2^2 = 4$, and $\lambda_3 = 2^e = 8$. Using informal inductive reasoning, we can hypothesize that $\lambda_h = 2^h$. Once again, we can apply mathematical induction to formally prove this hypothesis:
- **Basis Step:** To prove $\lambda_0 = 2^0$
  - Proof: 1. $\lambda_0 = 1 = 2^0$
- **Inductive Hypothesis:** Assume $\lambda_h = 2^h$
- **Inductive Step**: To prove $\lambda_{h+1} = 2^{h+1}$
  - Proof: 1. $\lambda_{h+1} = 2 \times \lambda_h$ from the recursive definition 2. $= 2 \times 2^h$ by substitution using the inductive hypothesis 3. $= 2^{h+1}$
- Therefore the number of leaves in a complete binary tree of height h is given by:

$$\lambda = 2^h \quad \textbf{QED}$$

- The importance of this result is that it places a cap on the number of leaves that can occur in a binary tree, with that cap determined by the height of the tree. In effect, for a binary tree of height h, there can be at most 2^h leaves. To illustrate the significance, consider the problem of growing a binary tree; that is, adding new vertices to it. There are two ways to do this:

1. Add a new vertex as the child of a leaf, making the leaf into an internal vertex.

Binary Trees

- The number of leaves in the complete binary trees of heights 1, 2, and 3 are: $\lambda_1 = 2^1 = 2$, $\lambda_2 = 2^2 = 4$, and $\lambda_3 = 2^e = 8$. Using informal inductive reasoning, we can hypothesize that $\lambda_h = 2^h$. Once again, we can apply mathematical induction to formally prove this hypothesis:
- **Basis Step:** To prove $\lambda_0 = 2^0$
  - Proof: 1. $\lambda_0 = 1 = 2^0$
- **Inductive Hypothesis:** Assume $\lambda_h = 2^h$
- **Inductive Step**: To prove $\lambda_{h+1} = 2^{h+1}$
  - Proof: 1. $\lambda_{h+1} = 2 \times \lambda_h$ from the recursive definition 2. $= 2 \times 2^h$ by substitution using the inductive hypothesis 3. $= 2^{h+1}$
- Therefore the number of leaves in a complete binary tree of height h is given by:

$$\lambda = 2^h \quad \textbf{QED}$$

- The importance of this result is that it places a cap on the number of leaves that can occur in a binary tree, with that cap determined by the height of the tree. In effect, for a binary tree of height h, there can be at most 2^h leaves. To illustrate the significance, consider the problem of growing a binary tree; that is, adding new vertices to it. There are two ways to do this:

1. Add a new vertex as the child of a leaf, making the leaf into an internal vertex.
2. Add a second child to the children of an existing internal node.

Binary Trees

- The number of leaves in the complete binary trees of heights 1, 2, and 3 are: $\lambda_1 = 2^1 = 2$, $\lambda_2 = 2^2 = 4$, and $\lambda_3 = 2^e = 8$. Using informal inductive reasoning, we can hypothesize that $\lambda_h = 2^h$. Once again, we can apply mathematical induction to formally prove this hypothesis:
- **Basis Step:** To prove $\lambda_0 = 2^0$
  - Proof: 1. $\lambda_0 = 1 = 2^0$
- **Inductive Hypothesis:** Assume $\lambda_h = 2^h$
- **Inductive Step**: To prove $\lambda_{h+1} = 2^{h+1}$
  - Proof: 1. $\lambda_{h+1} = 2 \times \lambda_h$ from the recursive definition 2. $= 2 \times 2^h$ by substitution using the inductive hypothesis 3. $= 2^{h+1}$
- Therefore the number of leaves in a complete binary tree of height h is given by:

$$\lambda = 2^h \quad \textbf{QED}$$

- The importance of this result is that it places a cap on the number of leaves that can occur in a binary tree, with that cap determined by the height of the tree. In effect, for a binary tree of height h, there can be at most 2^h leaves. To illustrate the significance, consider the problem of growing a binary tree; that is, adding new vertices to it. There are two ways to do this:

1. Add a new vertex as the child of a leaf, making the leaf into an internal vertex.
2. Add a second child to the children of an existing internal node.

- Now if we also wish to preserve the height of the tree, only leaves that are less than a distance h from the root can be assigned children. But a complete binary tree has no such leaves and all its internal vertices have exactly 2 children. Therefore, among all full binary trees of the same height, the complete binary tree has the maximum number of internal vertices and therefore the maximum number of leaves .

Binary Trees

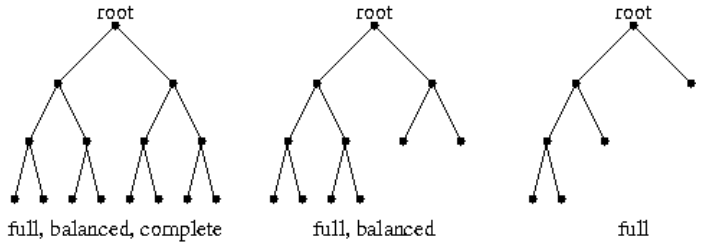
- The number of leaves in the complete binary trees of heights 1, 2, and 3 are: $\lambda_1 = 2^1 = 2$, $\lambda_2 = 2^2 = 4$, and $\lambda_3 = 2^e = 8$. Using informal inductive reasoning, we can hypothesize that $\lambda_h = 2^h$. Once again, we can apply mathematical induction to formally prove this hypothesis:
- **Basis Step:** To prove $\lambda_0 = 2^0$
  - Proof: 1. $\lambda_0 = 1 = 2^0$
- **Inductive Hypothesis:** Assume $\lambda_h = 2^h$
- **Inductive Step**: To prove $\lambda_{h+1} = 2^{h+1}$
  - Proof: 1. $\lambda_{h+1} = 2 \times \lambda_h$ from the recursive definition 2. $= 2 \times 2^h$ by substitution using the inductive hypothesis 3. $= 2^{h+1}$
- Therefore the number of leaves in a complete binary tree of height h is given by:

$$\lambda = 2^h \quad \textbf{QED}$$

- The importance of this result is that it places a cap on the number of leaves that can occur in a binary tree, with that cap determined by the height of the tree. In effect, for a binary tree of height h, there can be at most 2^h leaves. To illustrate the significance, consider the problem of growing a binary tree; that is, adding new vertices to it. There are two ways to do this:

1. Add a new vertex as the child of a leaf, making the leaf into an internal vertex.
2. Add a second child to the children of an existing internal node.

- Now if we also wish to preserve the height of the tree, only leaves that are less than a distance h from the root can be assigned children. But a complete binary tree has no such leaves and all its internal vertices have exactly 2 children. Therefore, among all full binary trees of the same height, the complete binary tree has the maximum number of internal vertices and therefore the maximum number of leaves .
- The following example shows three full binary trees; however, only the left-most tree is complete.



Binary Trees

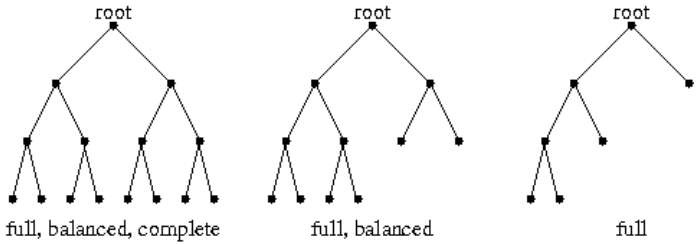
- The number of leaves in the complete binary trees of heights 1, 2, and 3 are: $\lambda_1 = 2^1 = 2$, $\lambda_2 = 2^2 = 4$, and $\lambda_3 = 2^e = 8$. Using informal inductive reasoning, we can hypothesize that $\lambda_h = 2^h$. Once again, we can apply mathematical induction to formally prove this hypothesis:
- **Basis Step:** To prove $\lambda_0 = 2^0$
  - Proof: 1. $\lambda_0 = 1 = 2^0$
- **Inductive Hypothesis:** Assume $\lambda_h = 2^h$
- **Inductive Step**: To prove $\lambda_{h+1} = 2^{h+1}$
  - Proof: 1. $\lambda_{h+1} = 2 \times \lambda_h$ from the recursive definition 2. $= 2 \times 2^h$ by substitution using the inductive hypothesis 3. $= 2^{h+1}$
- Therefore the number of leaves in a complete binary tree of height h is given by:

$$\lambda = 2^h \quad \textbf{QED}$$

- The importance of this result is that it places a cap on the number of leaves that can occur in a binary tree, with that cap determined by the height of the tree. In effect, for a binary tree of height h, there can be at most 2^h leaves. To illustrate the significance, consider the problem of growing a binary tree; that is, adding new vertices to it. There are two ways to do this:

1. Add a new vertex as the child of a leaf, making the leaf into an internal vertex.
2. Add a second child to the children of an existing internal node.

- Now if we also wish to preserve the height of the tree, only leaves that are less than a distance h from the root can be assigned children. But a complete binary tree has no such leaves and all its internal vertices have exactly 2 children. Therefore, among all full binary trees of the same height, the complete binary tree has the maximum number of internal vertices and therefore the maximum number of leaves .
- The following example shows three full binary trees; however, only the left-most tree is complete.



- Therefore no vertices can be added to the left binary tree; four vertices can be added to the centre binary tree; and eight vertices can be added to the right binary tree without the height of the tree being increased. As a result, except for the complete binary tree, for all other binary trees of height h:

$$\lambda < 2^h$$

FULL

Theorem: Every binary tree has an odd number of vertices.

Theorem: Every binary tree has an odd number of vertices.

HINT: Use the Handshaking Lemma, that “Every graph has an even number of vertices of odd degree” (from the Graph Theory PPT)

Theorem: Every binary tree has an odd number of vertices.

Proof Apart from the root, every vertex in a binary tree is of odd degree. We know that there are even number of such odd vertices. Therefore when the root (which is of even degree) is added to this number, the total number of vertices is odd.

Proof of the Lemma

Proposition: Every graph has an even number of vertices of odd degree.

Proof:

The sum of all the degrees is equal to twice the number of edges. Since the sum of the degrees is even and the sum of the degrees of vertices with even degree is even, the sum of the degrees of vertices with odd degree must be even. If the sum of the degrees of vertices with odd degree is even, there must be an even number of those vertices.

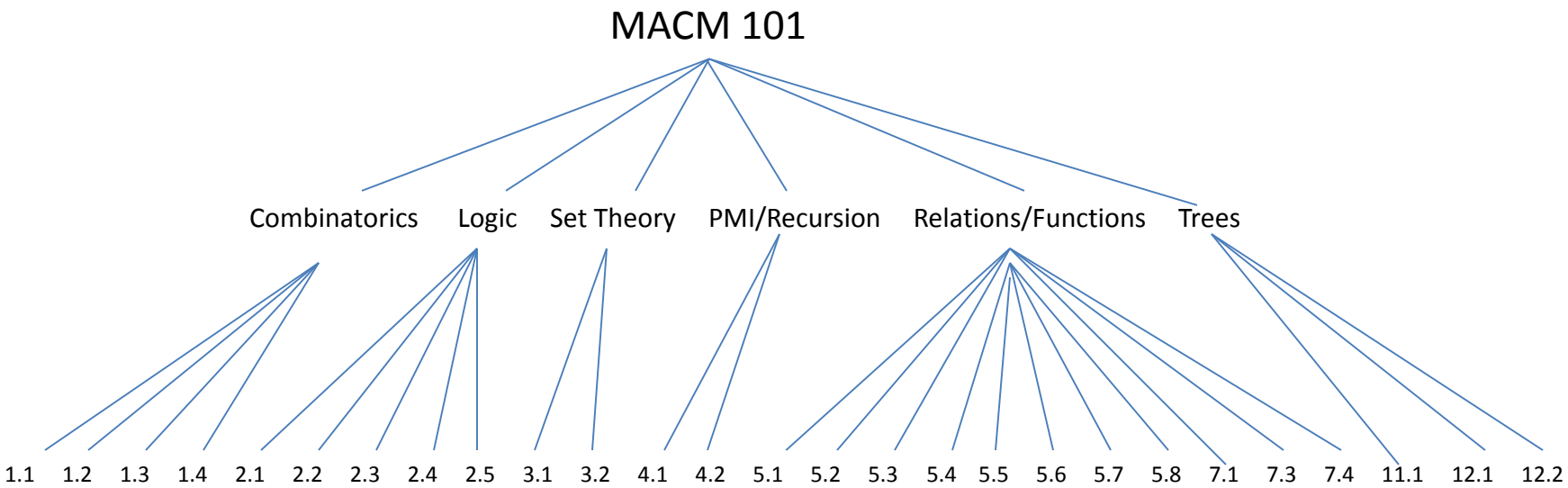
QED

Application of Trees: Tree Searching

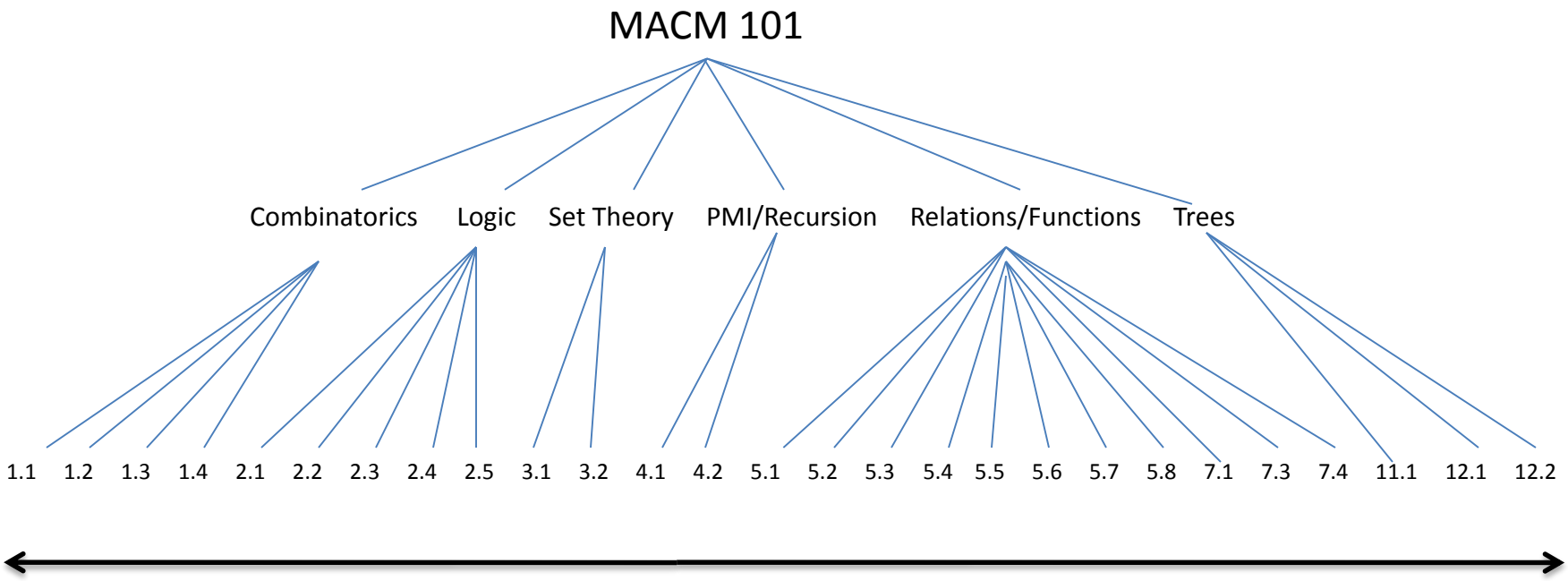
There are two well-known search methods.

- They are known as depth-first search(DFS) and breadth-first search (BFS).
- Each of these methods lists the vertices as they are encountered, and indicates the direction in which each edge is first traversed.
- The methods differ only in the way in which the vertex-lists are constructed.

Example: Studying for the Final

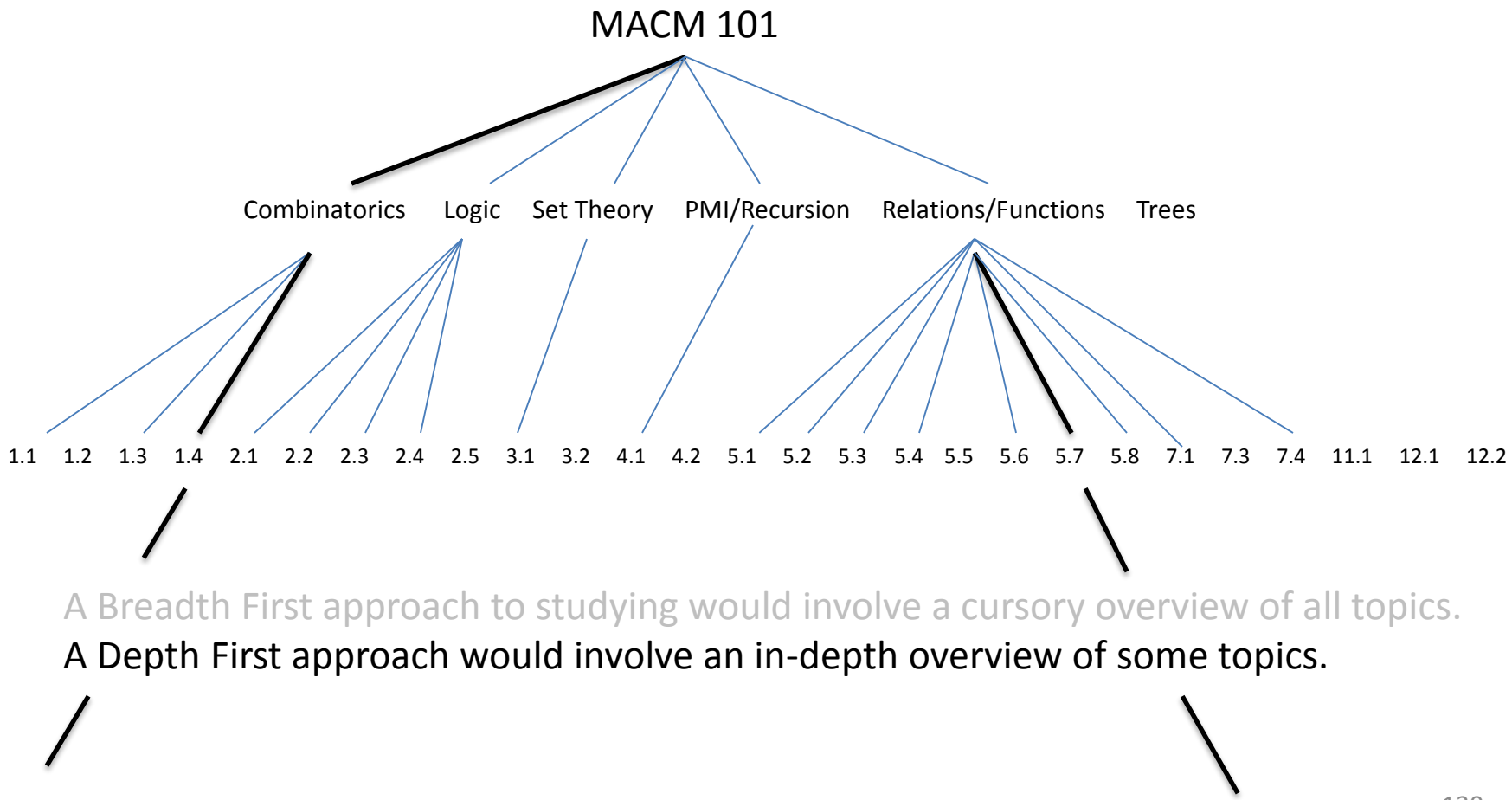


Example: Studying for the Final

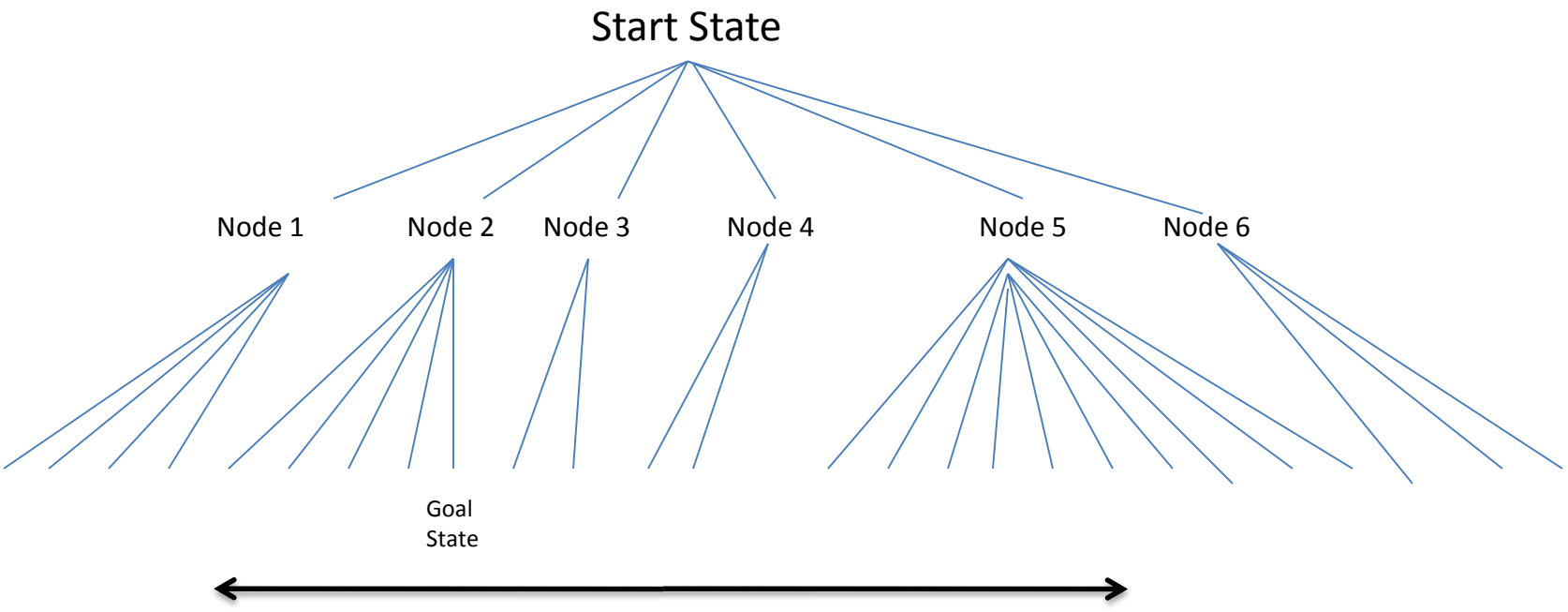


A Breadth First approach to studying would involve a cursory overview of all topics.

Example: Studying for the Final

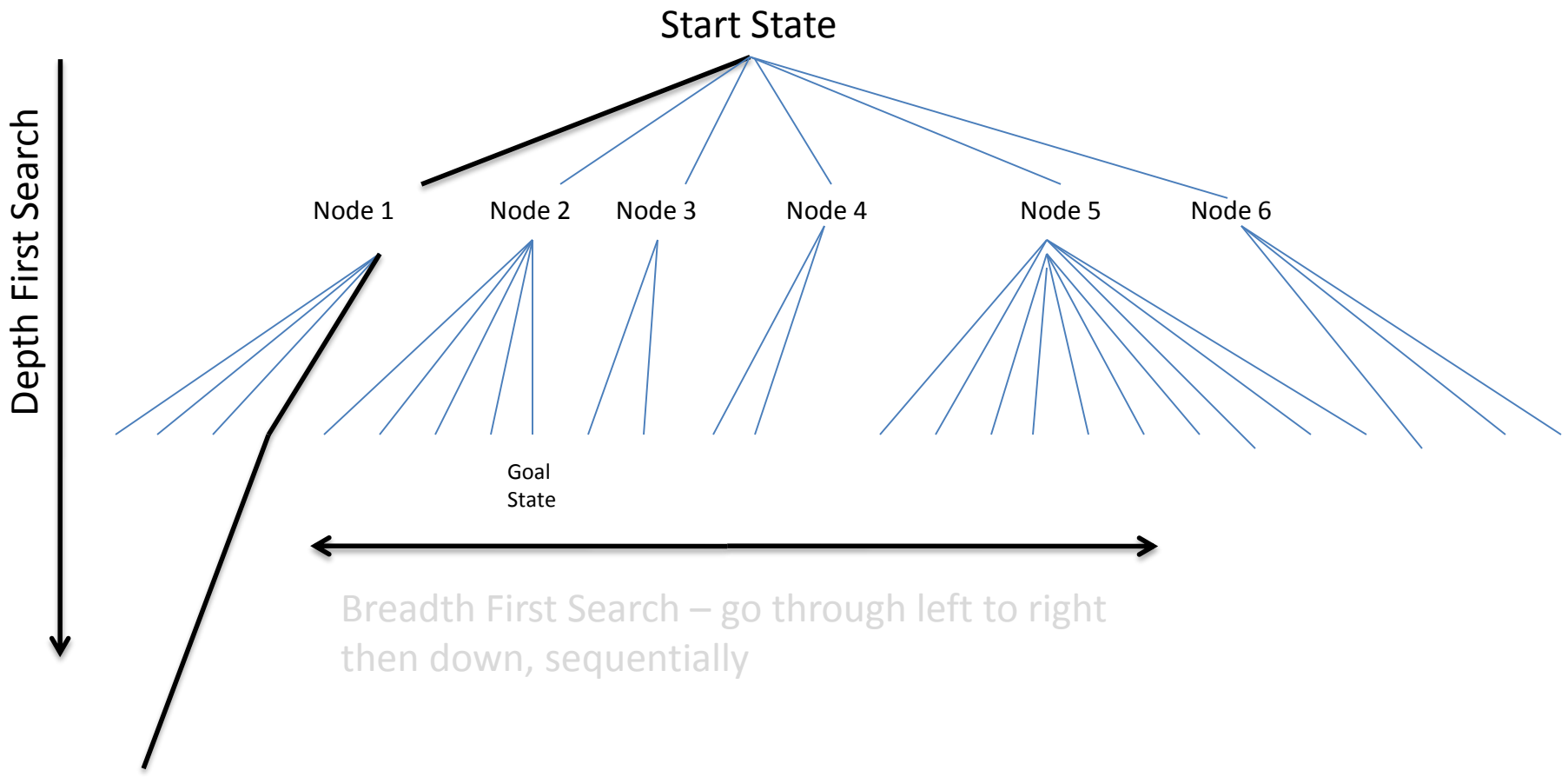


Example: Search Trees



Breadth First Search – go through left to right then down, sequentially

Example: Search Trees



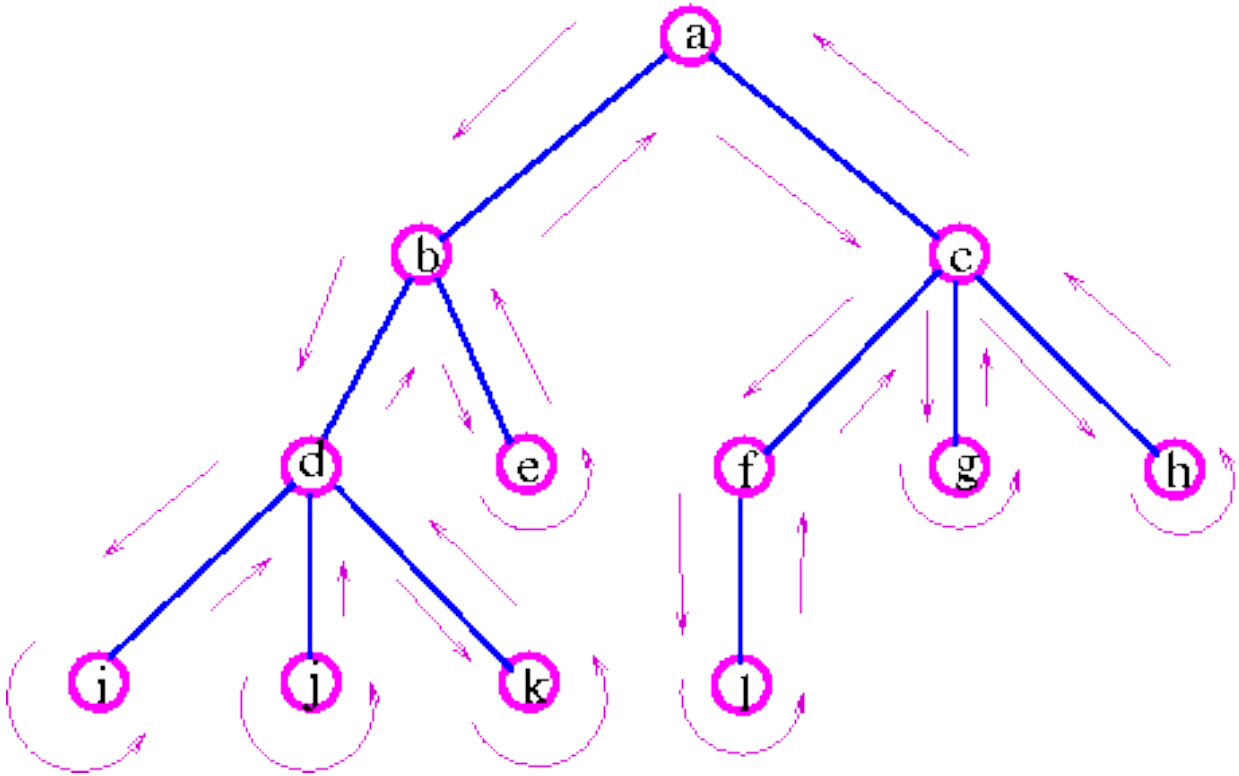
Which Search Strategy?

- Depends on *a priori* knowledge – heuristics.

Which Search Strategy?

- Depends on *a priori* knowledge – heuristics.
- You should do both for this final examination – know everything in-depth.

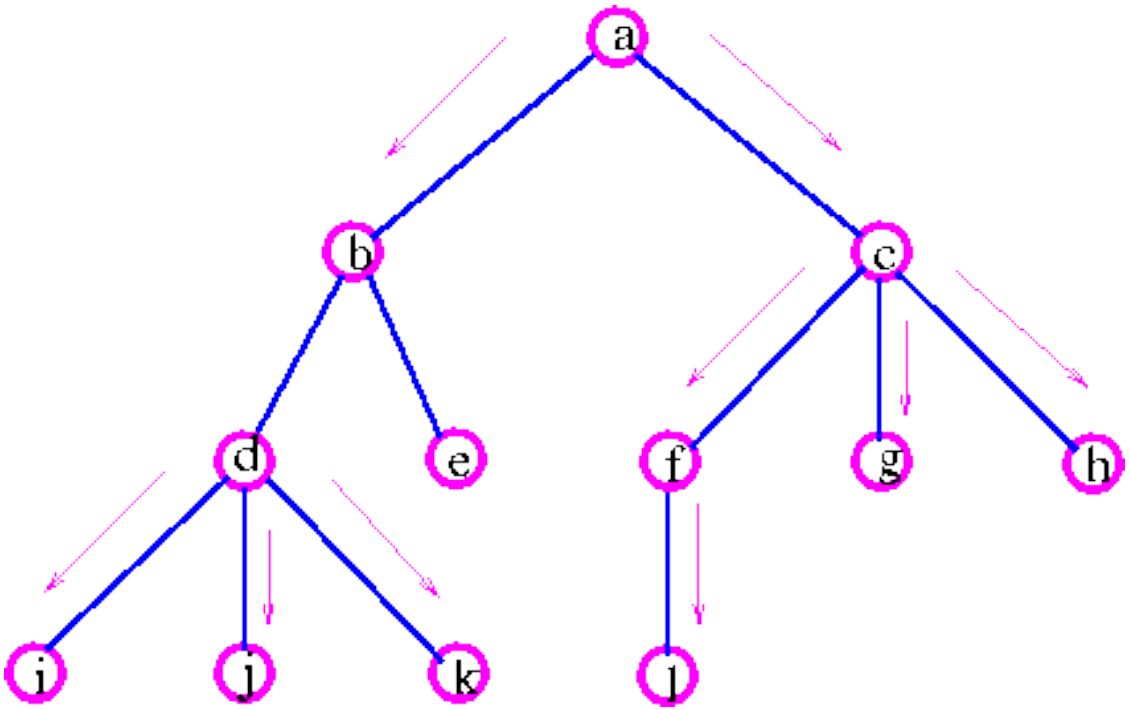
DFS



DFS

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

BFS



BFS

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

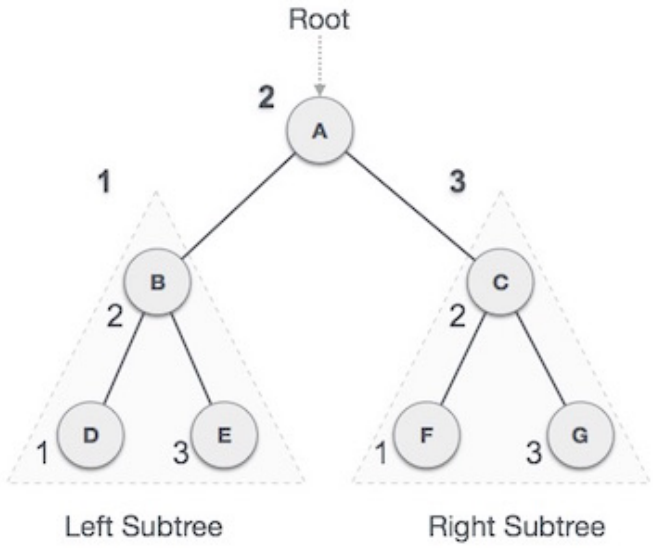
Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

Tree Traversal

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

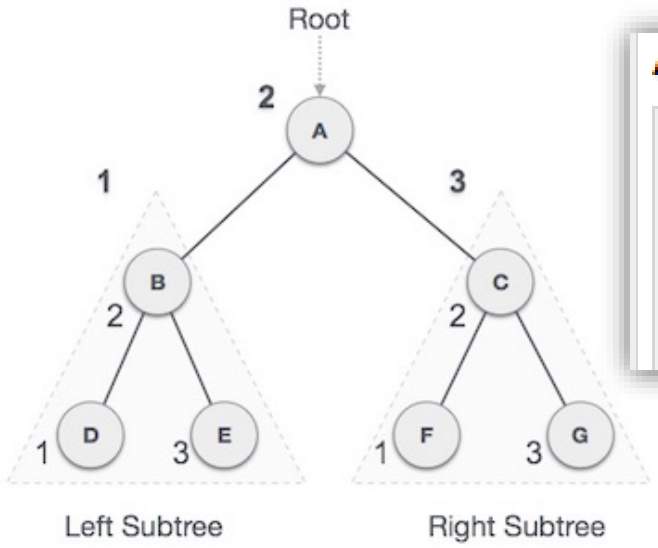
$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Tree Traversal

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



Algorithm

```
Until all nodes are traversed -
Step 1 - Recursively traverse left subtree.
Step 2 - Visit root node.
Step 3 - Recursively traverse right subtree.
```

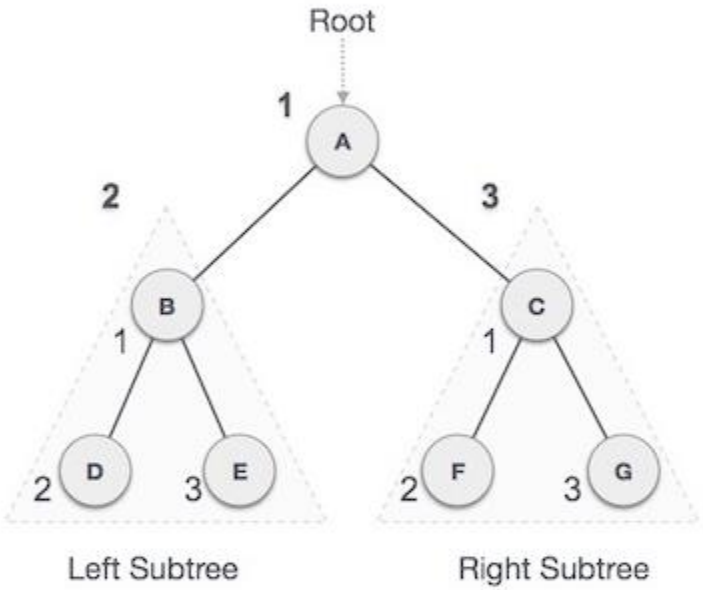
We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Tree Traversal

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



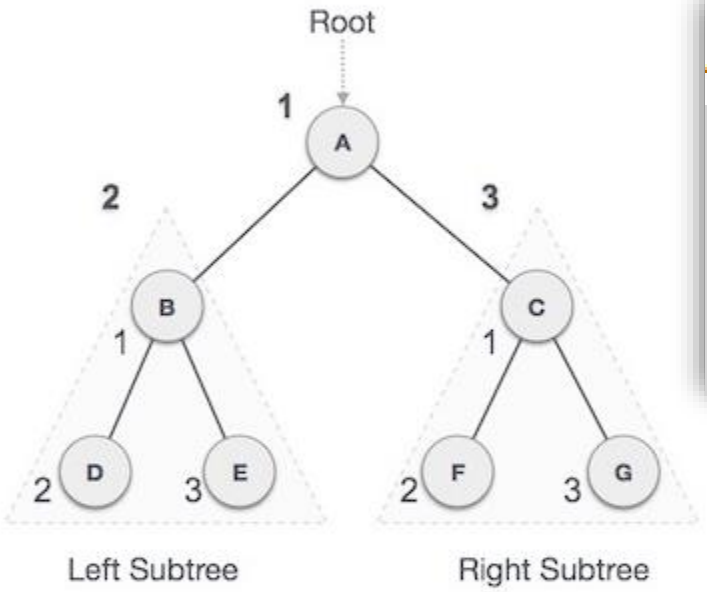
We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Tree Traversal

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



Algorithm

```
Until all nodes are traversed -
Step 1 - Visit root node.
Step 2 - Recursively traverse left subtree.
Step 3 - Recursively traverse right subtree.
```

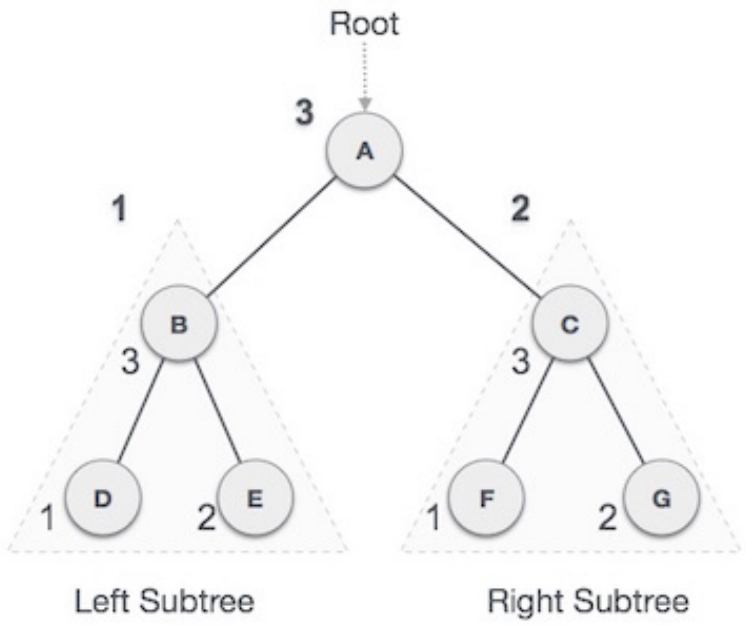
We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Tree Traversal

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



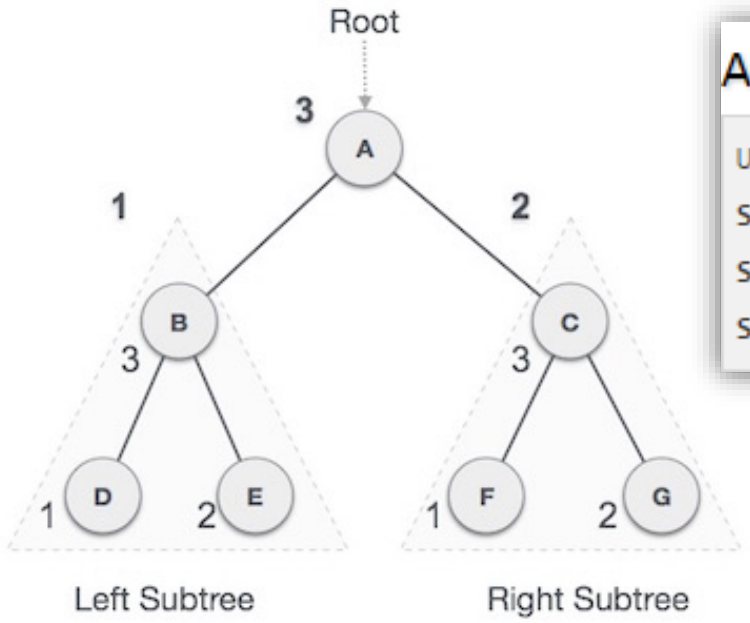
We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Tree Traversal

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



Algorithm

```
Until all nodes are traversed -
Step 1 - Recursively traverse left subtree.
Step 2 - Recursively traverse right subtree.
Step 3 - Visit root node.
```

We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Binary Search Trees (BSTs) as Data Structures

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as –

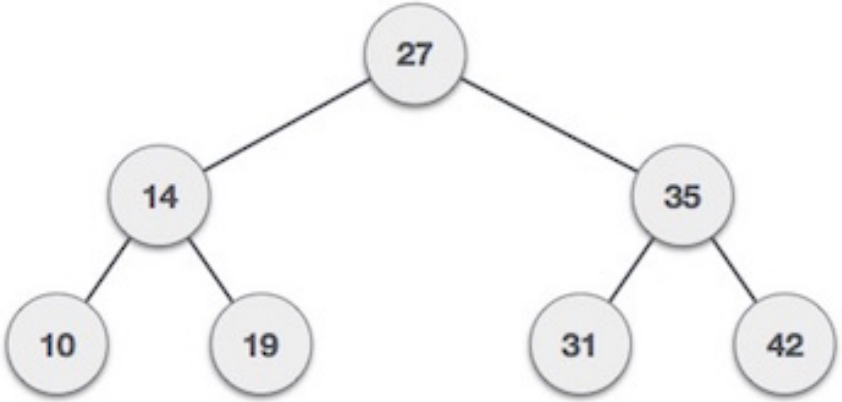
```
left_subtree (keys)  ≤  node (key)  ≤  right_subtree (keys)
```

Binary Search Trees (BSTs) as Data Structures

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST –



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Binary Search Trees (BSTs) as Data Structures

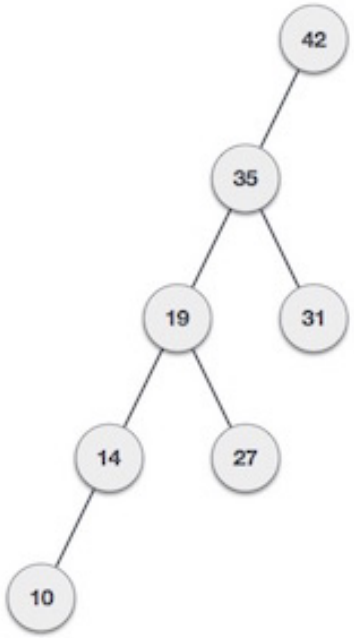
Basic Operations

Following are the basic operations of a tree –

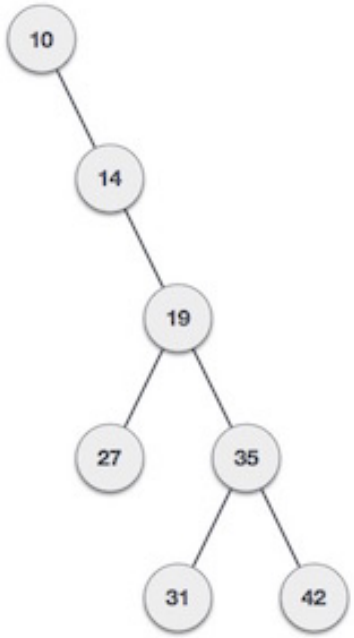
- **Search** – Searches an element in a tree.
- **Insert** – Inserts an element in a tree.
- **Pre-order Traversal** – Traverses a tree in a pre-order manner.
- **In-order Traversal** – Traverses a tree in an in-order manner.
- **Post-order Traversal** – Traverses a tree in a post-order manner.

Question Regarding BSTs

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this –



If input 'appears' non-increasing manner



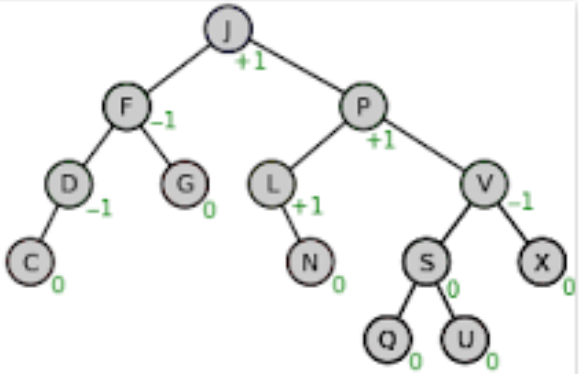
If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Definition

In computer science, an **AVL tree** is a self-balancing binary search **tree**. It was the first such data structure to be invented. In an **AVL tree**, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.

AVL tree - Wikipedia
https://en.wikipedia.org/wiki/AVL_tree



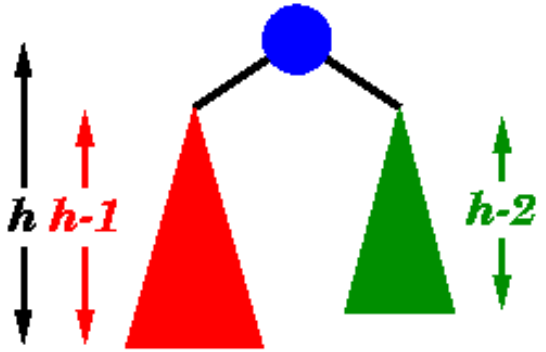
An **AVL tree** is another balanced binary search tree. Named after their inventors, **A**delson-**V**elskii and **L**andis, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an ***O(logn)*** search time. Addition and deletion operations also take ***O(logn)*** time.

Definition of an AVL tree

An AVL tree is a binary search tree which has the following properties:

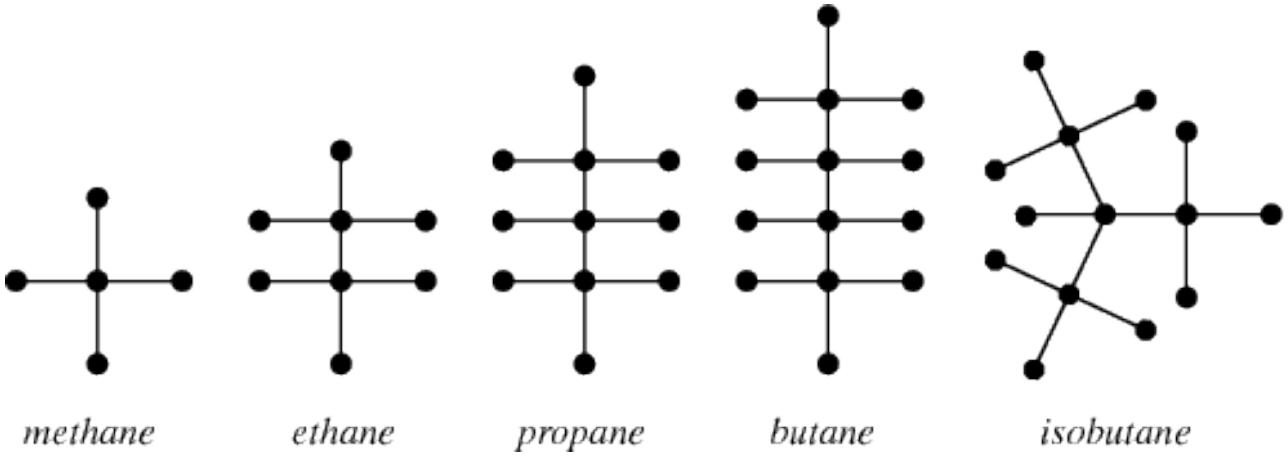
1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.

Balance requirement for an AVL tree: the left and right sub-trees differ by at most 1 in height.



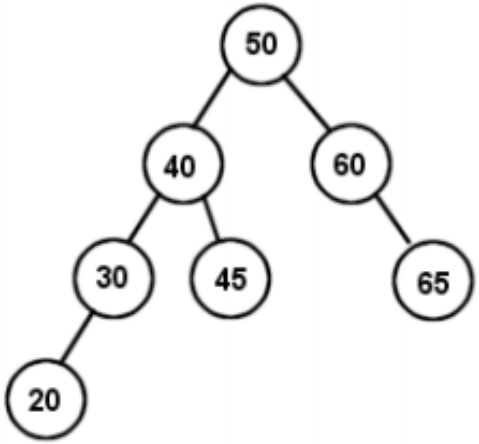
https://www.cs.auckland.ac.nz/software/AlgAnim/AVL.html

Application of Trees: Biochemistry



The enumeration of saturated hydrocarbons (saturation just refers to the fact that each carbon has its maximum number of bonds to hydrogen).

Application of Trees: Sorting



Create tree via:

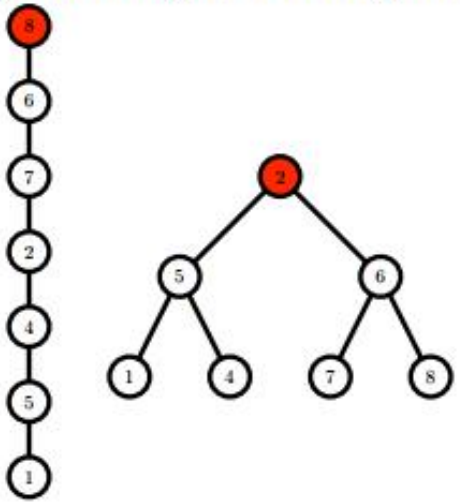
- *First number is the root.*
- *Put number in tree by traversing to an end vertex*
  - *If number less than or equal vertex number go left branch*
  - *If number greater than vertex number go right branch*

Tree above for the sequence: 50 40 60 30 20 65 45

Application of Trees: Data Structures

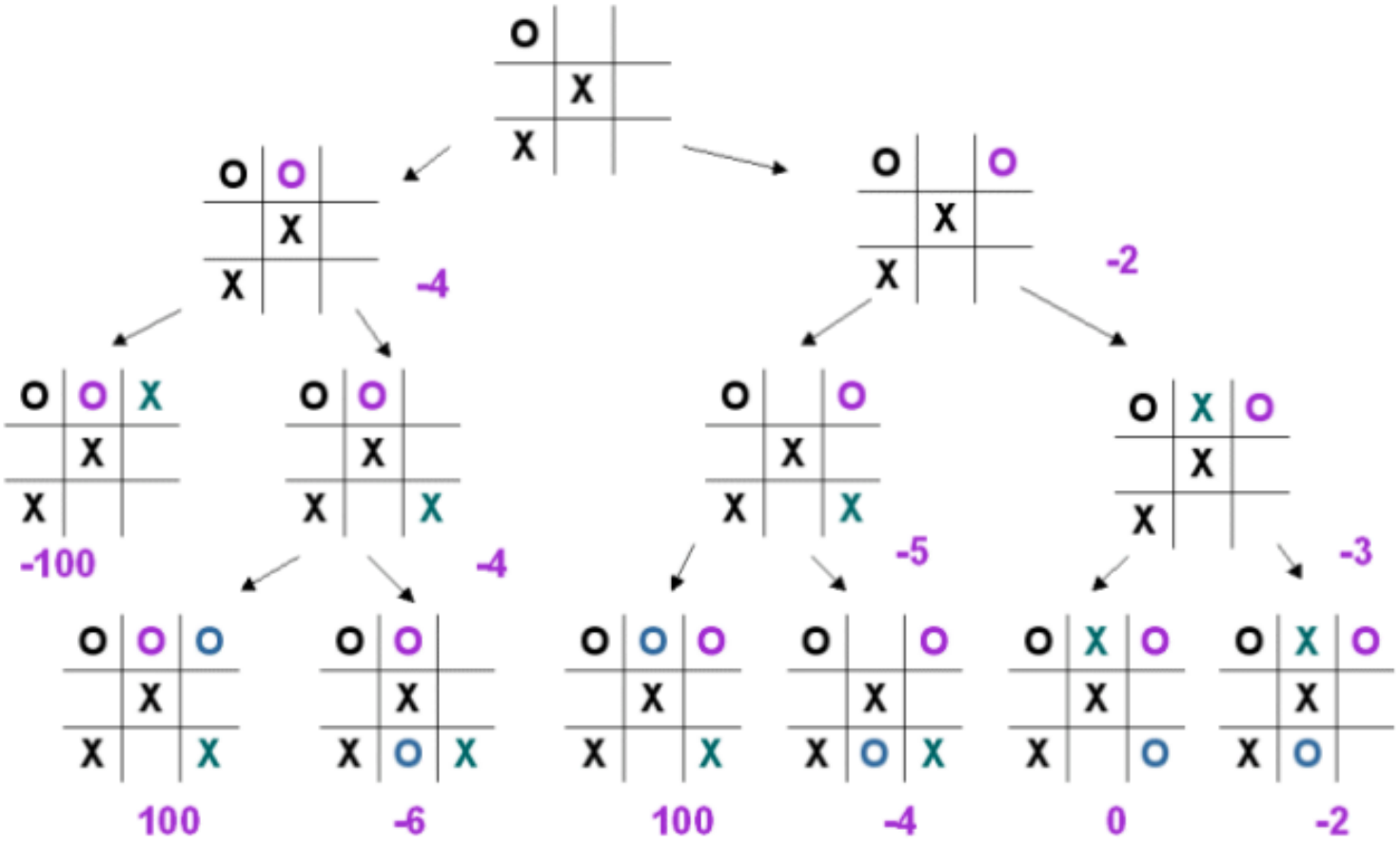
Rooted trees can be used to store data in a computer's memory in many different ways.

Consider a list of seven numbers $1, 5, 4, 2, 7, 6, 8$. *The following trees show two ways of storing this data, as a binary tree and as a stack.*



Both trees are rooted trees and both representations have their advantages. However it is important in both cases to know the starting point of the data, i.e. **the root**.

Application of Trees: Games



Presentation Terminated