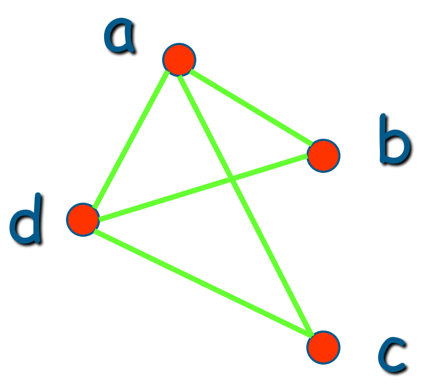
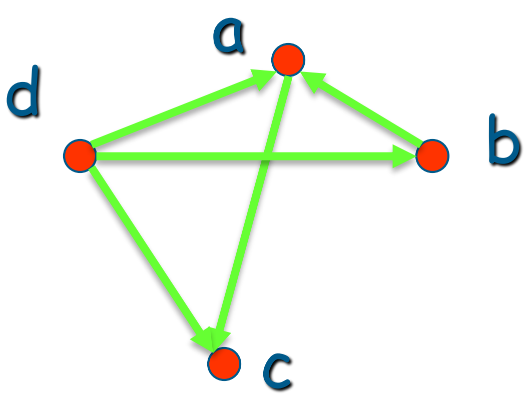
We will cover these parts of the book (8th edition):

10.3
10.4.1-10.4.3
10.4.5
11.1.1-11.1.3 (up to page 788)
11.2.1, 11.2.2
11.4.1
11.5
12.1
12.2.1
12.3.1

Representing Graphs (Adjacency list)





Vertex	Adjacent Vertices
a	b, c, d
b	a, d
c	a, d
d	a, b, c

Initial Vertex	Terminal Vertices
a	c
b	a
c	
d	a, b, c

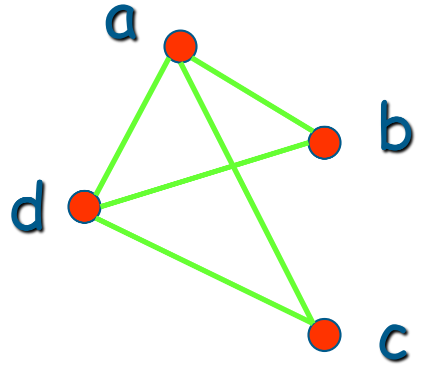
Representing Graphs

- **Definition:** Let G = (V, E) be a simple graph with |V| = n. Suppose that the vertices of G are listed in arbitrary order as $v_1, v_2, \ldots, v_n$.
- The **adjacency matrix** A (or A_G) of G, with respect to this listing of the vertices, is the $n \times n$ zero-one matrix with 1 as its (i, j)th entry when v_i and v_j are adjacent, and 0 otherwise.
- In other words, for an adjacency matrix $A = [a_{ij}]$,
- $a_{ij} = 1$ if $\{v_i, v_j\}$ is an edge of G,

$a_{ij} = 0$ otherwise.

Representing Graphs

- **Example:** What is the adjacency matrix A_G for the following graph G based on the order of vertices a, b, c, d ?



Solution:

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

Note: Adjacency matrices of undirected graphs are always symmetric.

Representing Graphs

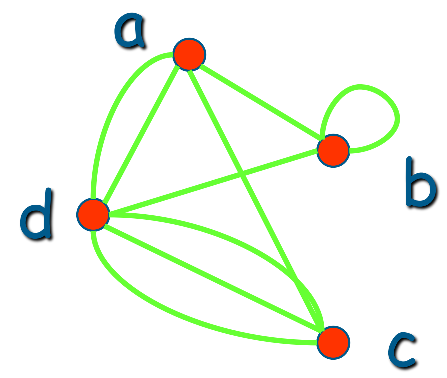
- For the representation of graphs with **multiple edges**, we can no longer use zero-one matrices.

- Instead, we use **matrices of natural numbers**.

- The (i, j)th entry of such a matrix equals the **number of edges** that are associated with $\{v_i, v_j\}$.

Representing Graphs

- **Example:** What is the adjacency matrix A_G for the following graph G based on the order of vertices a, b, c, d ?



Solution:

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

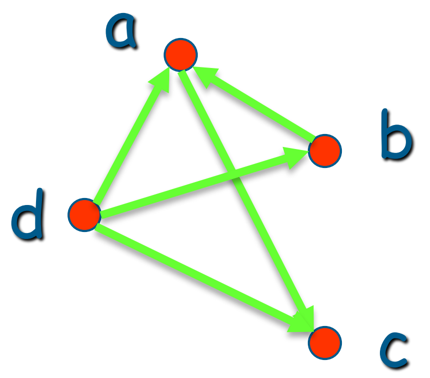
Note: For undirected graphs, adjacency matrices are symmetric.

Representing Graphs

- **Definition:** Let $G = (V, E)$ be a directed graph with $|V| = n$. Suppose that the vertices of G are listed in arbitrary order as $v_1, v_2, \ldots, v_n$.
- The **adjacency matrix** A (or A_G) of G, with respect to this listing of the vertices, is the $n \times n$ zero-one matrix with 1 as its (i, j)th entry when there is an edge from v_i to v_j, and 0 otherwise.
- In other words, for an adjacency matrix $A = [a_{ij}]$,
- $a_{ij} = 1$ if (v_i, v_j) is an edge of G,
$a_{ij} = 0$ otherwise.

Representing Graphs

- **Example:** What is the adjacency matrix A_G for the following graph G based on the order of vertices a, b, c, d ?



Solution:

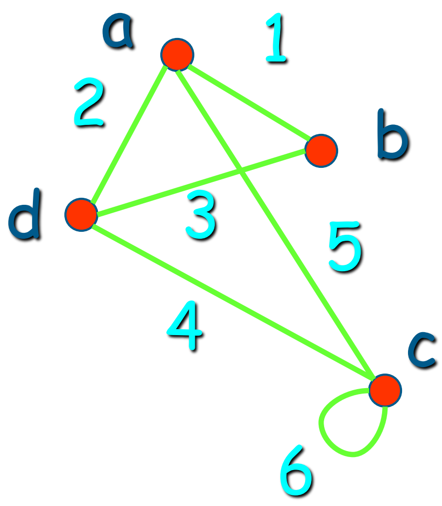
$$A_G = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Representing Graphs

- **Definition:** Let $G = (V, E)$ be an undirected graph with $|V| = n$ and $|E| = m$. Suppose that the vertices and edges of G are listed in arbitrary order as $v_1, v_2, \ldots, v_n$ and $e_1, e_2, \ldots, e_m$, respectively.
- The **incidence matrix** of G with respect to this listing of the vertices and edges is the $n \times m$ zero-one matrix with 1 as its (i, j)th entry when edge e_j is incident with vertex v_i, and 0 otherwise.
- In other words, for an incidence matrix $M = [m_{ij}]$,
- $m_{ij} = 1$ if edge e_j is incident with v_i
  $m_{ij} = 0$ otherwise.

Representing Graphs

- **Example:** What is the incidence matrix M for the following graph G based on the order of vertices a, b, c, d and edges 1, 2, 3, 4, 5, 6?



Solution:

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

Note: Incidence matrices of directed graphs contain two 1s per column for edges connecting two vertices and one 1 per column for loops.

Isomorphism of Graphs

- **Definition:** The simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are **isomorphic** if there is a bijection (an one-to-one and onto function) f from V_1 to V_2 with the property that a and b are adjacent in G_1 if and only if f(a) and f(b) are adjacent in G_2, for all a and b in V_1.
- Such a function f is called an **isomorphism**.
- In other words, G_1 and G_2 are isomorphic if their vertices can be ordered in such a way that the adjacency matrices M_{G_1} and M_{G_2} are identical.

Isomorphism of Graphs

- From a visual standpoint, G_1 and G_2 are isomorphic if they can be arranged in such a way that their **displays are identical** (of course without changing adjacency).
- Unfortunately, for two simple graphs, each with n vertices, there are **n! possible isomorphisms** that we have to check in order to show that these graphs are isomorphic.
- However, showing that two graphs are **not** isomorphic can be easy.

Isomorphism of Graphs

▸For this purpose we can check **invariants**, that is, properties that two isomorphic simple graphs must both have.

▸For example, they must have

- the same number of vertices,
- the same number of edges, and
- the same degrees of vertices.

▸Note that two graphs that **differ** in any of these invariants are not isomorphic, but two graphs that **match** in all of them are not necessarily isomorphic.

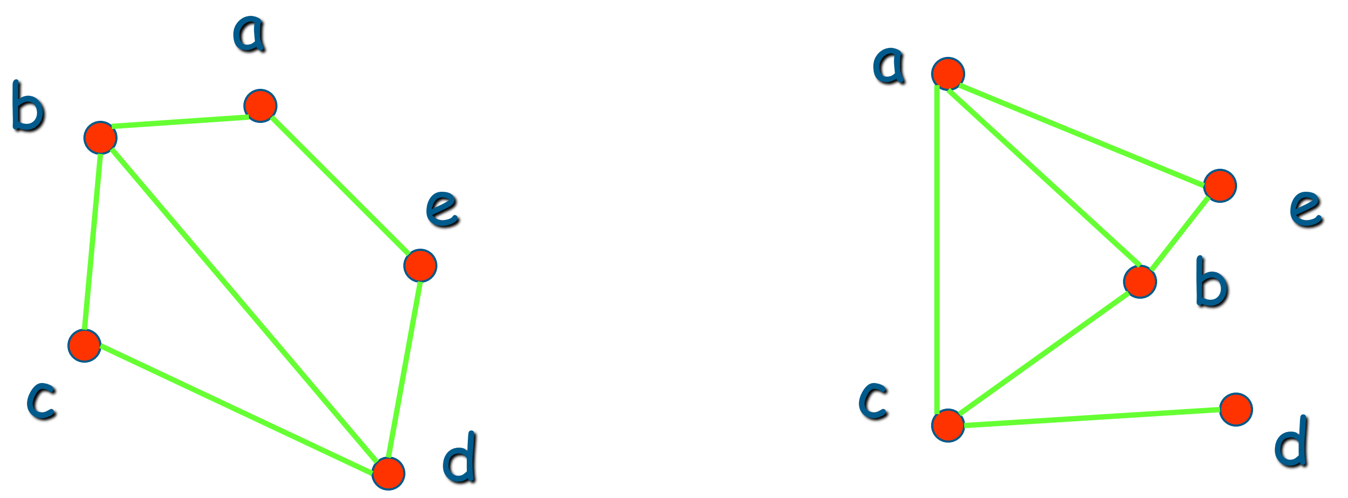
Isomorphism of Graphs

- **Example I:** Are the following two graphs isomorphic?

Solution: Yes, they are isomorphic, because they can be arranged to look identical. You can see this if in the right graph you move vertex b to the left of the edge {a, c}. Then the isomorphism f from the left to the right graph is: f(a) = e, f(b) = a,
f(c) = b, f(d) = c, f(e) = d.

Isomorphism of Graphs

- **Example II:** How about these two graphs?



Solution: No, they are not isomorphic, because they differ in the degrees of their vertices.

Vertex d in right graph is of degree one, but there is no such vertex in the left graph.

Connectivity

- **Definition:** A **path** of length n from u to v, where n is a positive integer, in an **undirected graph** is a sequence of edges $e_1, e_2, \ldots, e_n$ of the graph such that $f(e_1) = \{x_0, x_1\}$, $f(e_2) = \{x_1, x_2\}$, …, $f(e_n) = \{x_{n-1}, x_n\}$, where $x_0 = u$ and $x_n = v$.
- When the graph is simple, we denote this path by its **vertex sequence** $x_0, x_1, \ldots, x_n$, since it uniquely determines the path.
- The path is a **circuit** if it begins and ends at the same vertex, that is, if $u = v$.

Connectivity

- **Definition (continued):** The path or circuit is said to **pass through** or traverse $x_1, x_2, \ldots, x_{n-1}$.

- A path or circuit is **simple** if it does not contain the same edge more than once.

Connectivity

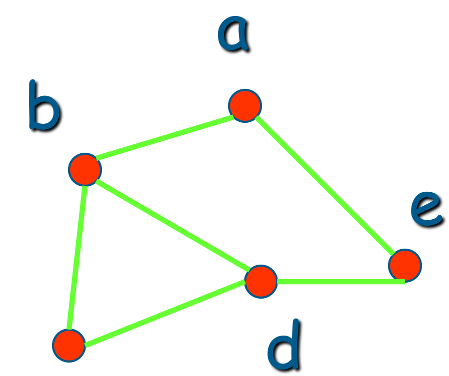
- **Definition:** A **path** of length n from u to v, where n is a positive integer, in a **directed multigraph** is a sequence of edges $e_1, e_2, \ldots, e_n$ of the graph such that $f(e_1) = (x_0, x_1)$, $f(e_2) = (x_1, x_2)$, …, $f(e_n) = (x_{n-1}, x_n)$, where $x_0 = u$ and $x_n = v$.
- When there are no multiple edges in the path, we denote this path by its **vertex sequence** $x_0, x_1, \ldots, x_n$, since it uniquely determines the path.
- The path is a **circuit** if it begins and ends at the same vertex, that is, if u = v.
- A path or circuit is called **simple** if it does not contain the same edge more than once.

Connectivity

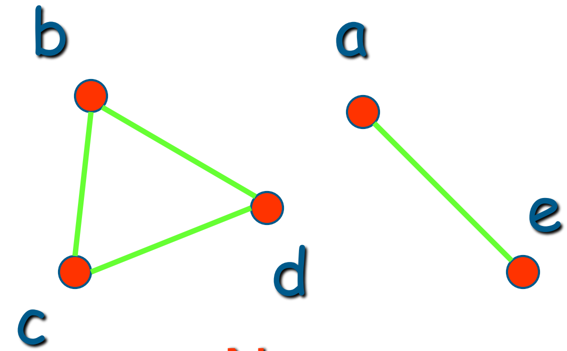
- Let us now look at something new:
- **Definition:** An undirected graph is called **connected** if there is a path between every pair of distinct vertices in the graph.
- For example, any two computers in a network can communicate if and only if the graph of this network is connected.
- **Note:** A graph consisting of only one vertex is always connected, because it does not contain any pair of distinct vertices.

Connectivity

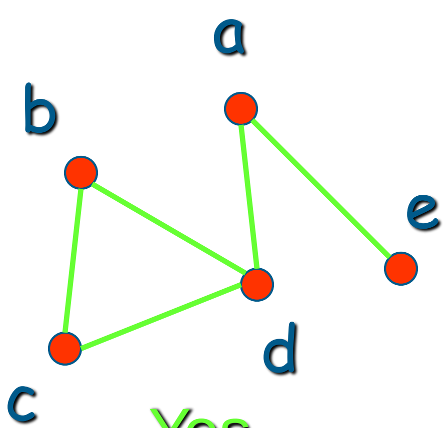
- **Example:** Are the following graphs connected?



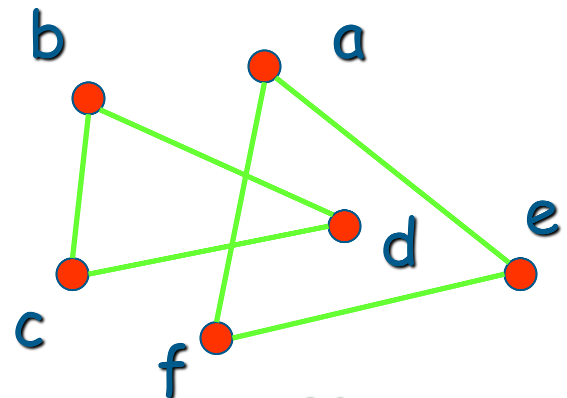
Yes



No



Yes



No

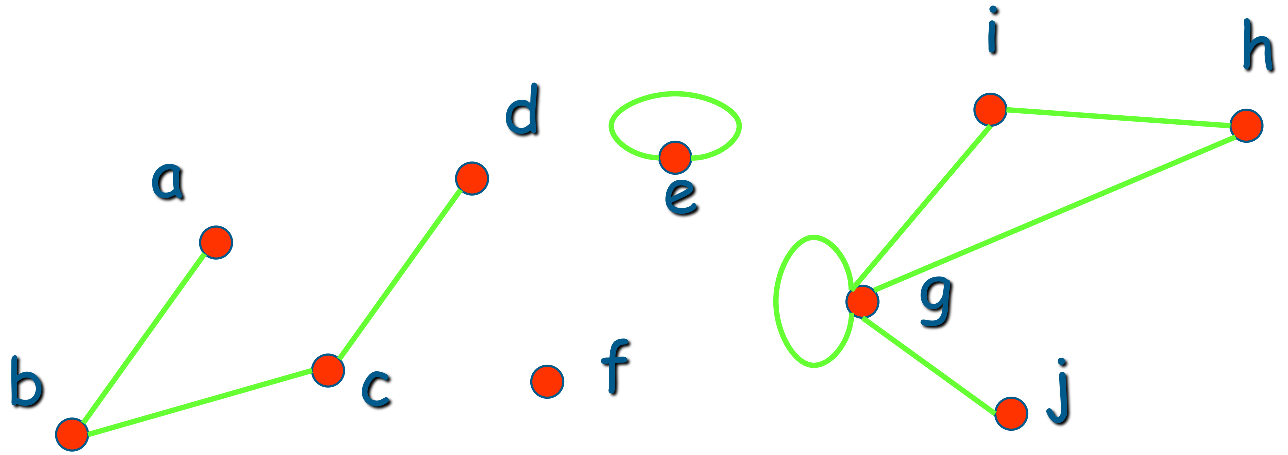
Connectivity

- **Theorem:** There is a **simple** path between every pair of distinct vertices of a connected undirected graph.

- **Definition:** A graph that is not connected is the union of two or more connected subgraphs, each pair of which has no vertex in common. These disjoint connected subgraphs are called the **connected components** of the graph.

Connectivity

- **Example:** What are the connected components in the following graph?



Solution: The connected components are the graphs with vertices {a, b, c, d}, {e}, {f}, {i, g, h, j}.

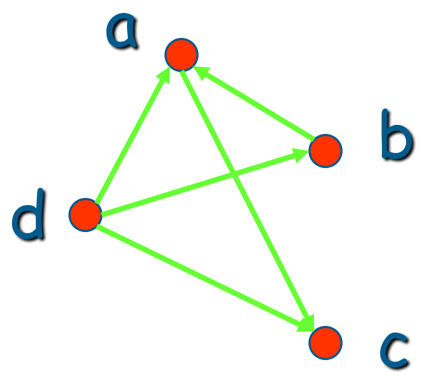
Connectivity

- **Definition:** A directed graph is **strongly connected** if there is a path from a to b and from b to a whenever a and b are vertices in the graph.

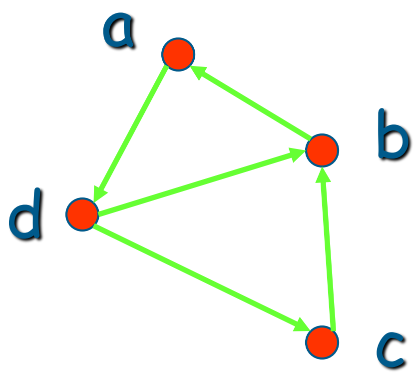
- **Definition:** A directed graph is **weakly connected** if there is a path between any two vertices in the underlying undirected graph.

Connectivity

- **Example:** Are the following directed graphs strongly or weakly connected?



Weakly connected, because, for example, there is no path from b to d.



Strongly connected, because there are paths between all possible pairs of vertices.

Connectivity

- **Idea:** The number and size of connected components and circuits are further invariants with respect to isomorphism of simple graphs.
- **Example:** Are these two graphs isomorphic?

Solution: No, because the right graph contains circuits of length 3, while the left graph does not.

- Trees

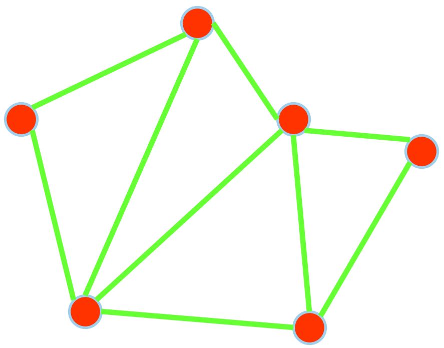
Trees

- **Definition:** A **tree** is a connected undirected graph with no simple circuits.
- Since a tree cannot have a simple circuit, a tree cannot contain multiple edges or loops.
- Therefore, any tree must be a **simple graph**.

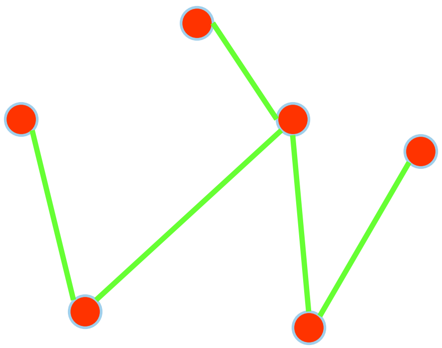
- **Theorem:** An undirected graph is a tree if and only if there is a **unique simple path** between any of its vertices.

Trees

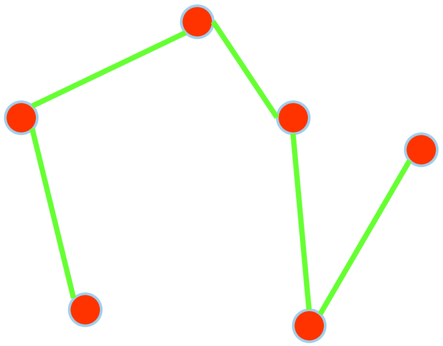
- **Example:** Are the following graphs trees?



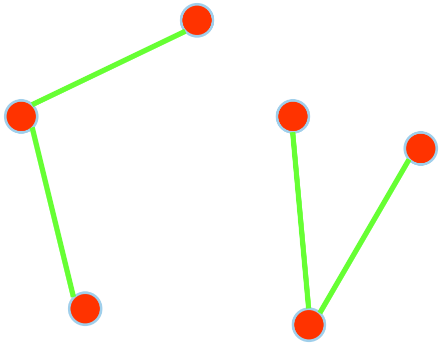
No.



Yes.



Yes.



No.

Trees

- **Definition:** An undirected graph that does not contain simple circuits and is not necessarily connected is called a **forest**.
- In general, we use trees to represent **hierarchical structures**.
- We often designate a particular vertex of a tree as the **root**. Since there is a unique path from the root to each vertex of the graph, we direct each edge away from the root.
- Thus, a tree together with its root produces a **directed graph** called a **rooted tree**.

Tree Terminology

- If v is a vertex in a rooted tree other than the root, the **parent** of v is the unique vertex u such that there is a directed edge from u to v.

- When u is the parent of v, v is called the **child** of u.

- Vertices with the same parent are called **siblings**.

- 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.

Tree Terminology

- The **descendants** of a vertex v are those vertices that have v as an ancestor.

- A vertex of a tree is called a **leaf** if it has no children.

- Vertices that have children are called **internal vertices**.

- If a is a vertex in a tree, then 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.

Tree Terminology

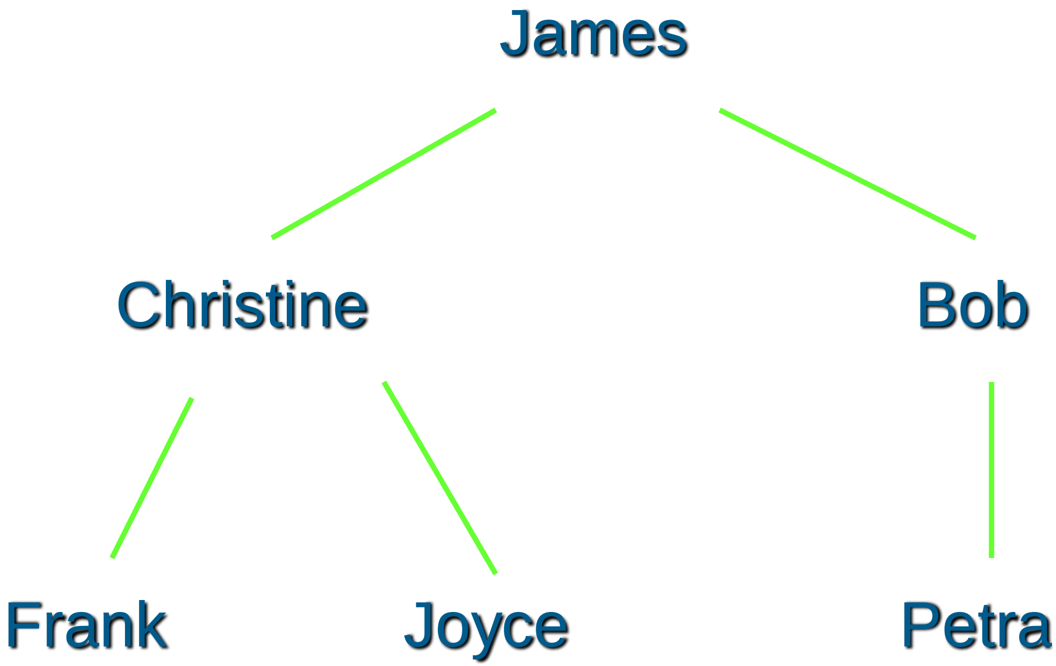
- The **level** of a vertex v in a rooted tree is the length of the unique path from the root to this vertex.

- The level of the root is defined to be zero.

- The **height** of a rooted tree is the maximum of the levels of vertices.

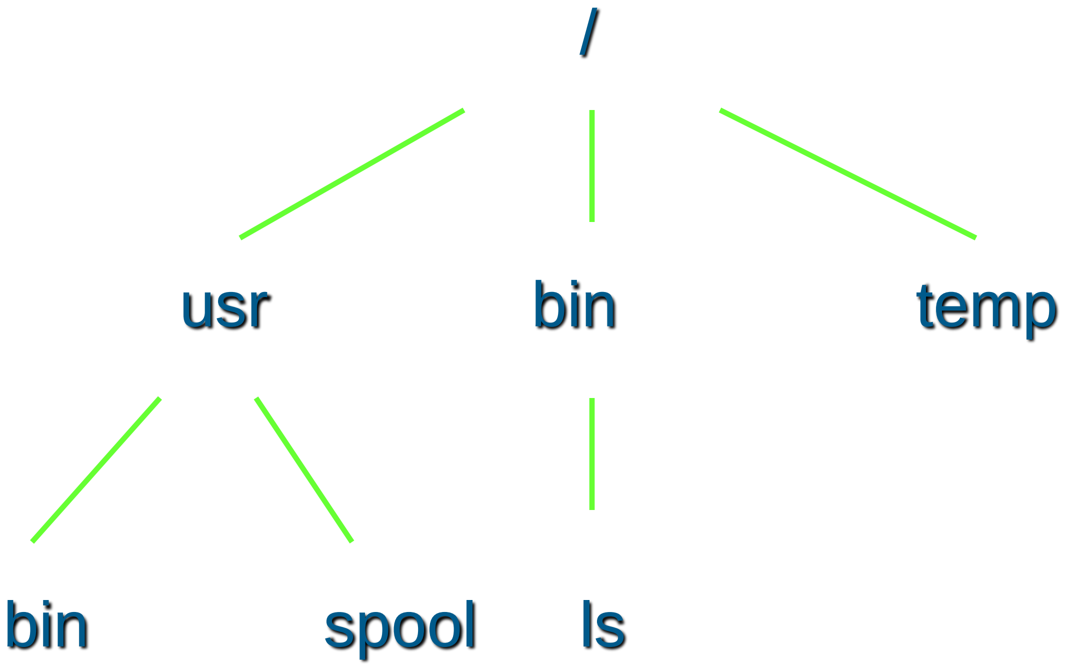
Trees

- **Example I:** Family tree



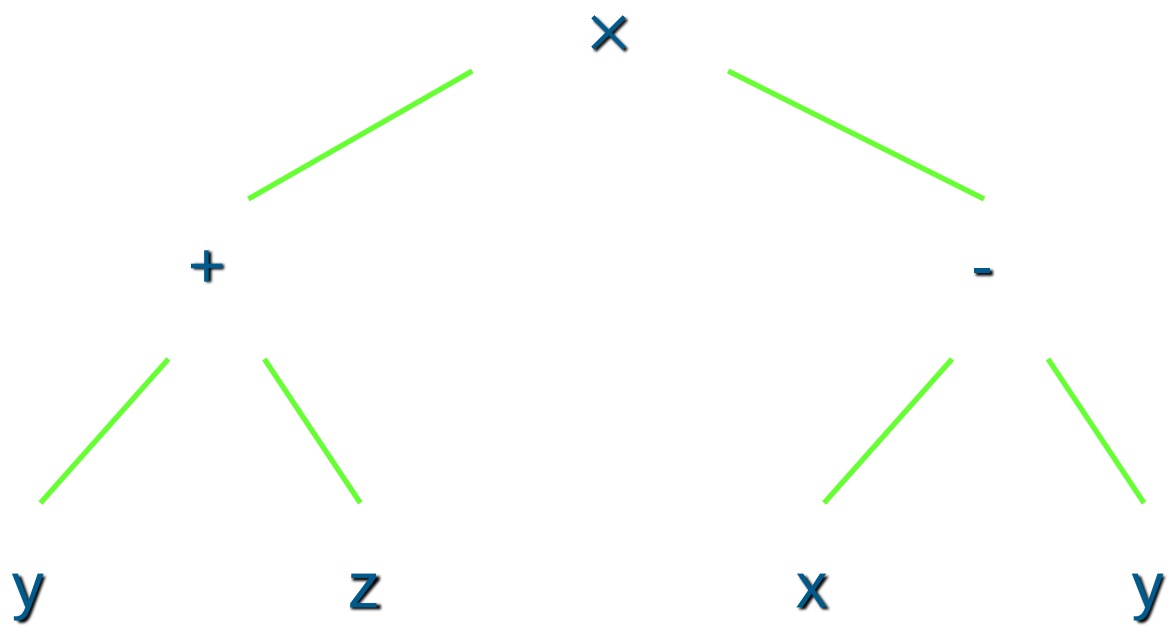
Trees

- **Example II:** File system



Trees

- **Example III:** Arithmetic expressions



This tree represents the expression (y + z) × (x - y).

Trees

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

- The tree is called a **full m-ary tree** if every internal vertex has exactly m children.

- An m-ary tree with m = 2 is called a **binary tree**.

- **Theorem:** A tree with n vertices has (n – 1) edges.

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

Binary Search Trees

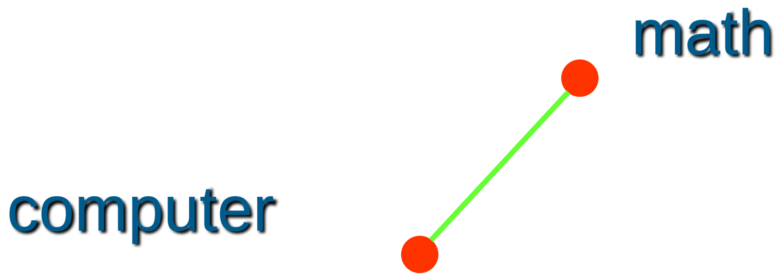
- If we want to perform a large number of searches in a particular list of items, it can be worthwhile to arrange these items in a **binary search tree** to facilitate the subsequent searches.
- A binary search tree is a binary tree in which each child of a vertex is designated as a **right or left child**, and each vertex is labeled with a **key**, which is one of the items.
- When we construct the tree, vertices are assigned keys so that the key of a vertex is both larger than the keys of all vertices in its left subtree and smaller than the keys of all vertices in its right subtree.

Binary Search Trees

- **Example:** Construct a binary search tree for the strings **math, computer, power, north, zoo, dentist, book**.

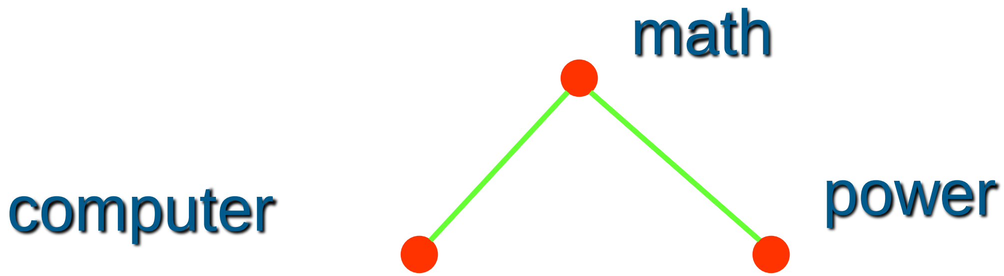
Binary Search Trees

- **Example:** Construct a binary search tree for the strings **math, computer, power, north, zoo, dentist, book**.



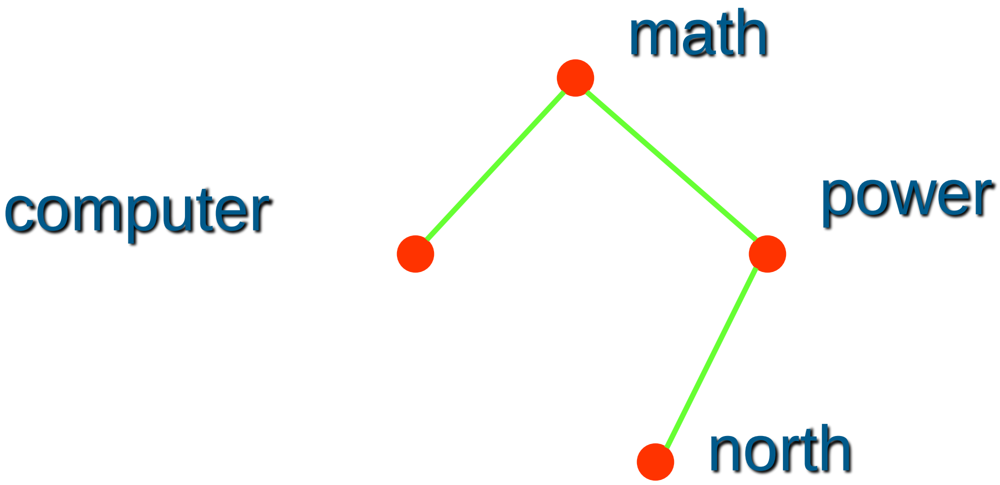
Binary Search Trees

- **Example:** Construct a binary search tree for the strings **math, computer, power, north, zoo, dentist, book**.



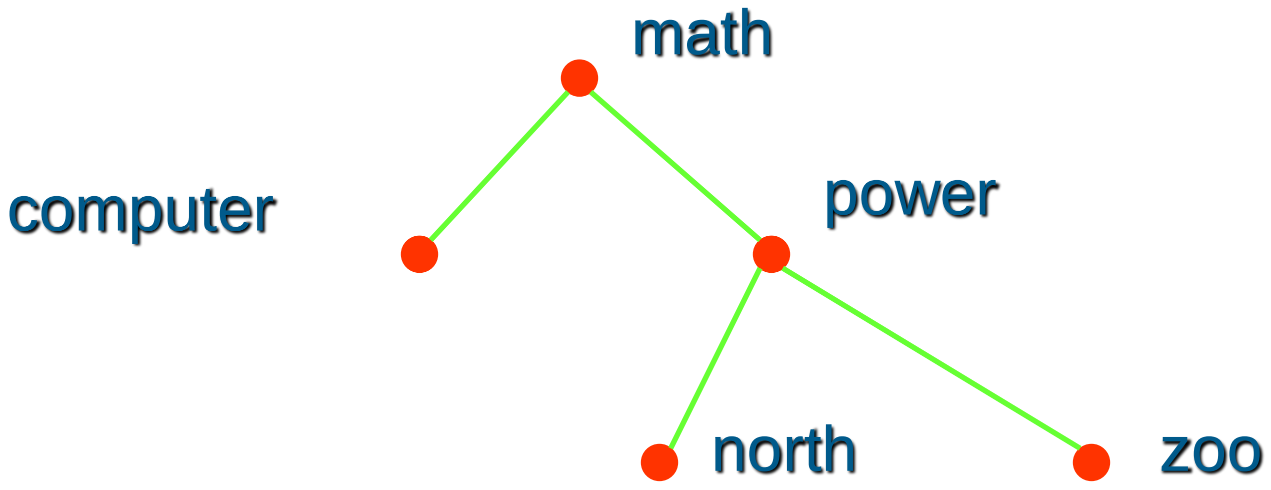
Binary Search Trees

- **Example:** Construct a binary search tree for the strings **math, computer, power, north, zoo, dentist, book**.



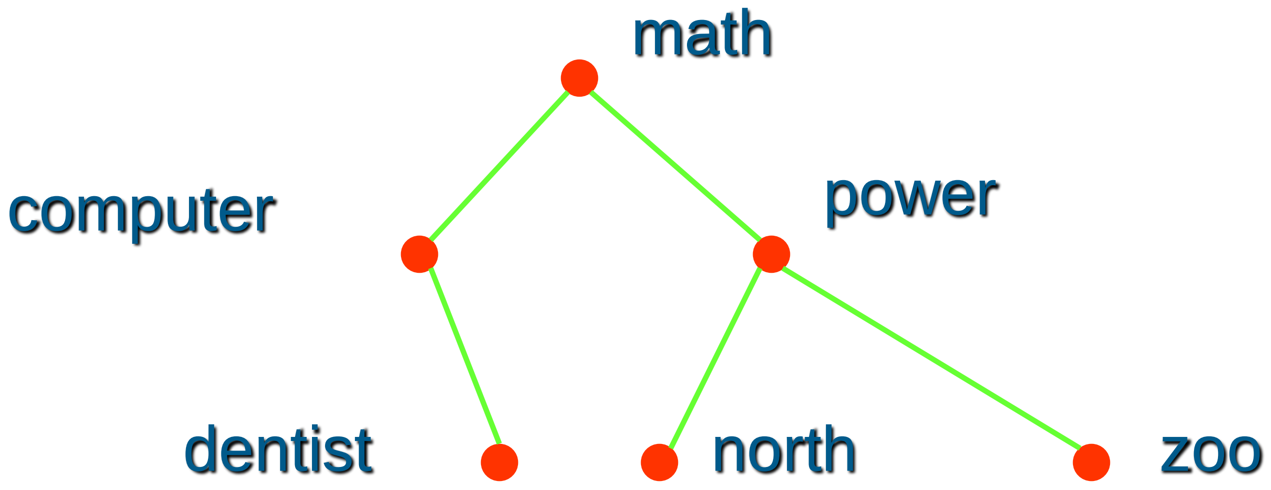
Binary Search Trees

- **Example:** Construct a binary search tree for the strings **math, computer, power, north, zoo, dentist, book**.



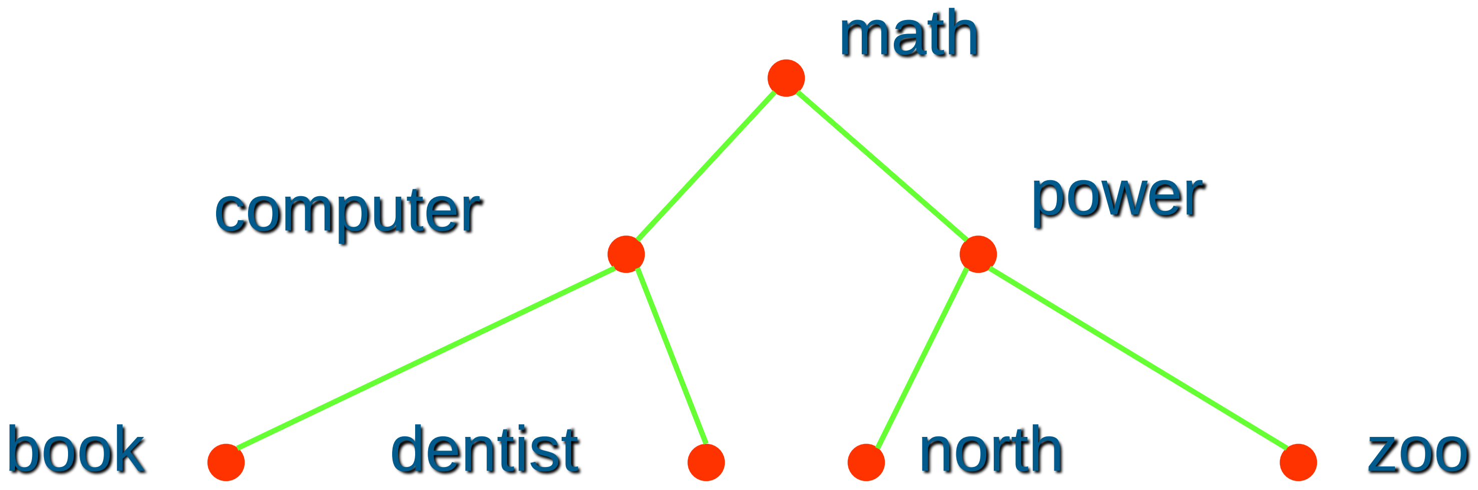
Binary Search Trees

- **Example:** Construct a binary search tree for the strings **math, computer, power, north, zoo, dentist, book**.



Binary Search Trees

- **Example:** Construct a binary search tree for the strings **math, computer, power, north, zoo, dentist, book**.



Binary Search Trees

- To perform a search in such a tree for an item x, we can start at the root and compare its key to x. If x is **less** than the key, we proceed to the **left** child of the current vertex, and if x is **greater** than the key, we proceed to the **right** one.

- This procedure is repeated until we either found the item we were looking for, or we cannot proceed any further.

- In a balanced tree representing a list of n items, search can be performed with a maximum of $\lceil \log(n + 1) \rceil$ steps (compare with binary search).

Binary Search Trees

ALGORITHM 1 Locating an Item in or Adding an Item to a Binary Search Tree.

procedure *insertion*(T: binary search tree,
x: item)
v := root of T
{a vertex not present in T has the value *null* }
while $v \neq null$ and $label(v) \neq x$
    if $x < label(v)$ **then**
        if left child of $v \neq null$ **then** v := left child of v
        else add *new vertex* as a left child of v and set $v := null$
    else
        if right child of $v \neq null$ **then** v := right child of v
        else add *new vertex* as a right child of v and set $v := null$
if root of $T = null$ **then** add a vertex v to the tree and label it with x
else if v is null or $label(v) \neq x$ **then** label *new vertex* with x and let v be this new vertex
return v {v = location of x}

Spanning Trees

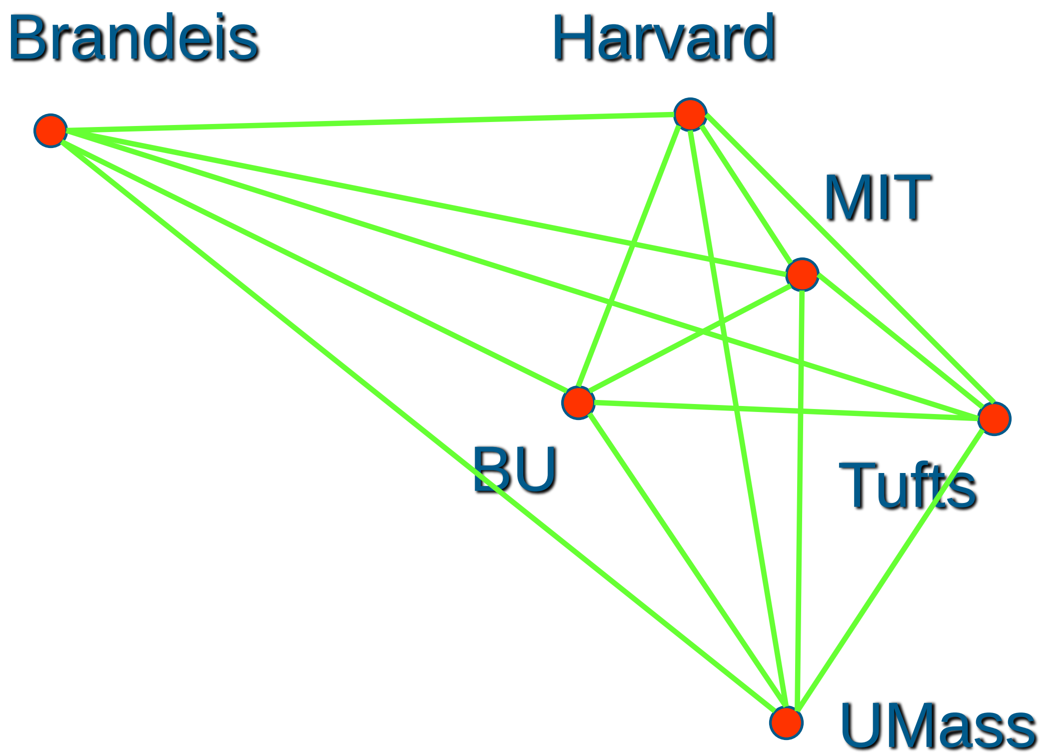
- **Definition:** Let G be a simple graph. A spanning tree of G is a subgraph of G that is a tree containing every vertex of G.

- **Note:** A spanning tree of G = (V, E) is a connected graph on V with a minimum number of edges (|V| - 1).

- **Example:** Since winters in Boston can be very cold, six universities in the Boston area decide to build a tunnel system that connects their libraries.

Spanning Trees

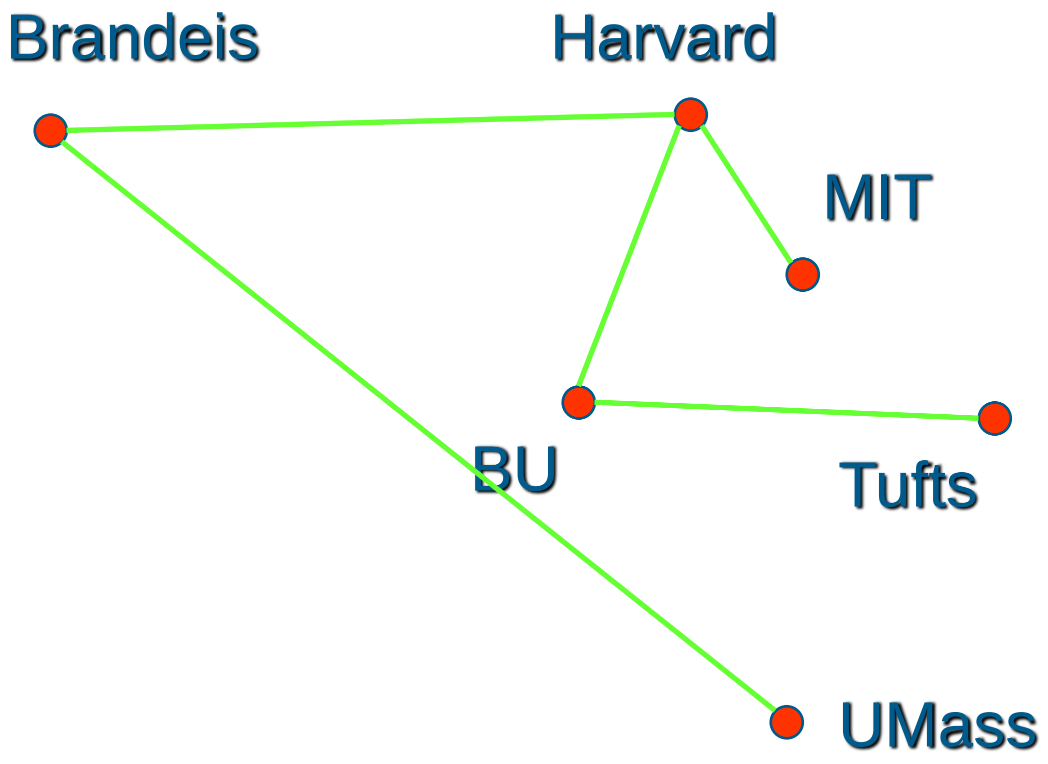
- The complete graph including all possible tunnels:



The spanning trees of this graph connect all libraries with a minimum number of tunnels.

Spanning Trees

- Example for a spanning tree:



Since there are 6 libraries, 5 tunnels are sufficient to connect all of them.

Spanning Trees

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

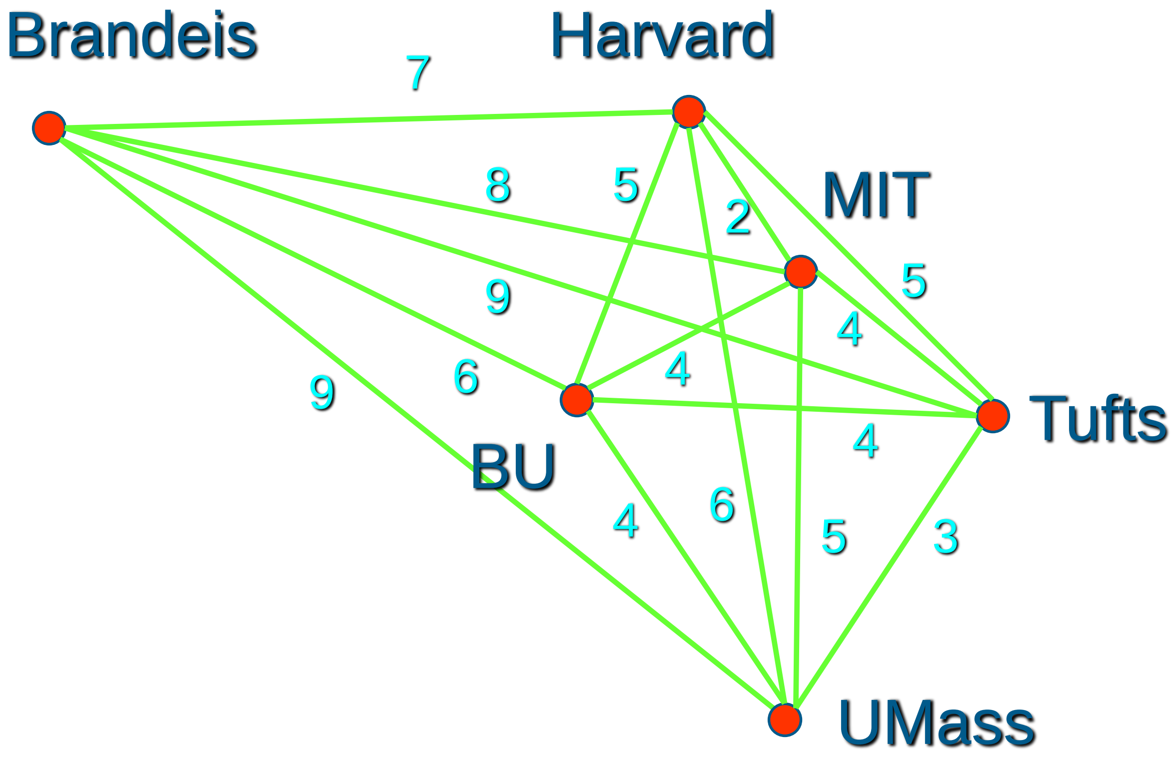
- Now imagine that you are in charge of the tunnel project. How can you determine a tunnel system of **minimal cost** that connects all libraries?

- **Definition:** A **minimum spanning tree** in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

- How can we find a minimum spanning tree?

Spanning Trees

- The complete graph with cost labels (in billion $):



The least expensive tunnel system costs $20 billion.

Spanning Trees

- **Prim's Algorithm:**

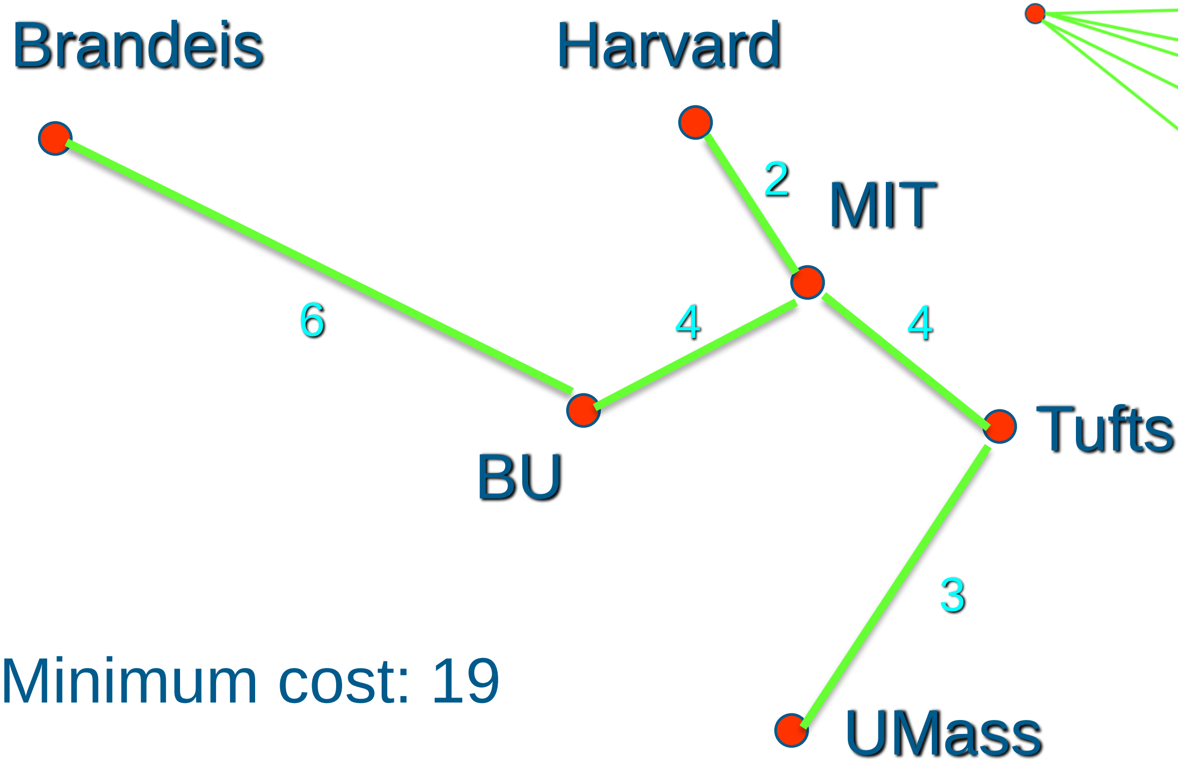
- Begin by choosing any edge with **smallest weight** and putting it into the spanning tree,

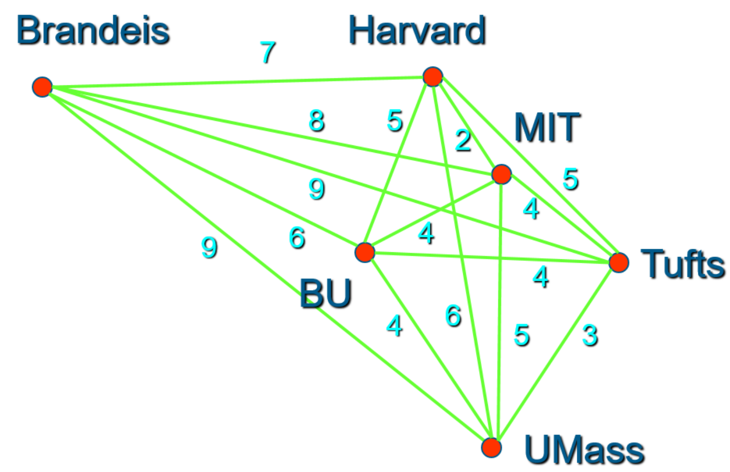
- successively add to the tree edges of **minimum weight** that are incident to a vertex already in the tree and not forming a simple circuit with those edges already in the tree,

- stop when (n – 1) edges have been added.

Spanning Trees

- **Example:** Use Prim's algorithm to design a minimum spanning tree for the libraries' graph.





Minimum cost: 19

Spanning Trees

- **Kruskal's Algorithm:**

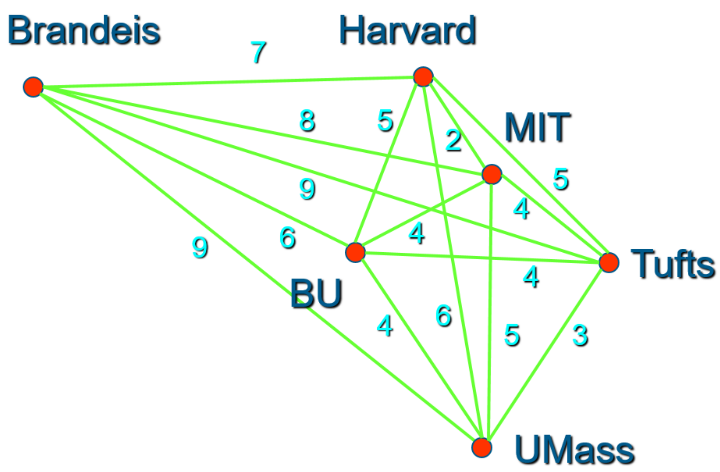
- Kruskal's algorithm is identical to Prim's algorithm, except that it does not demand new edges to be incident to a vertex already in the tree.

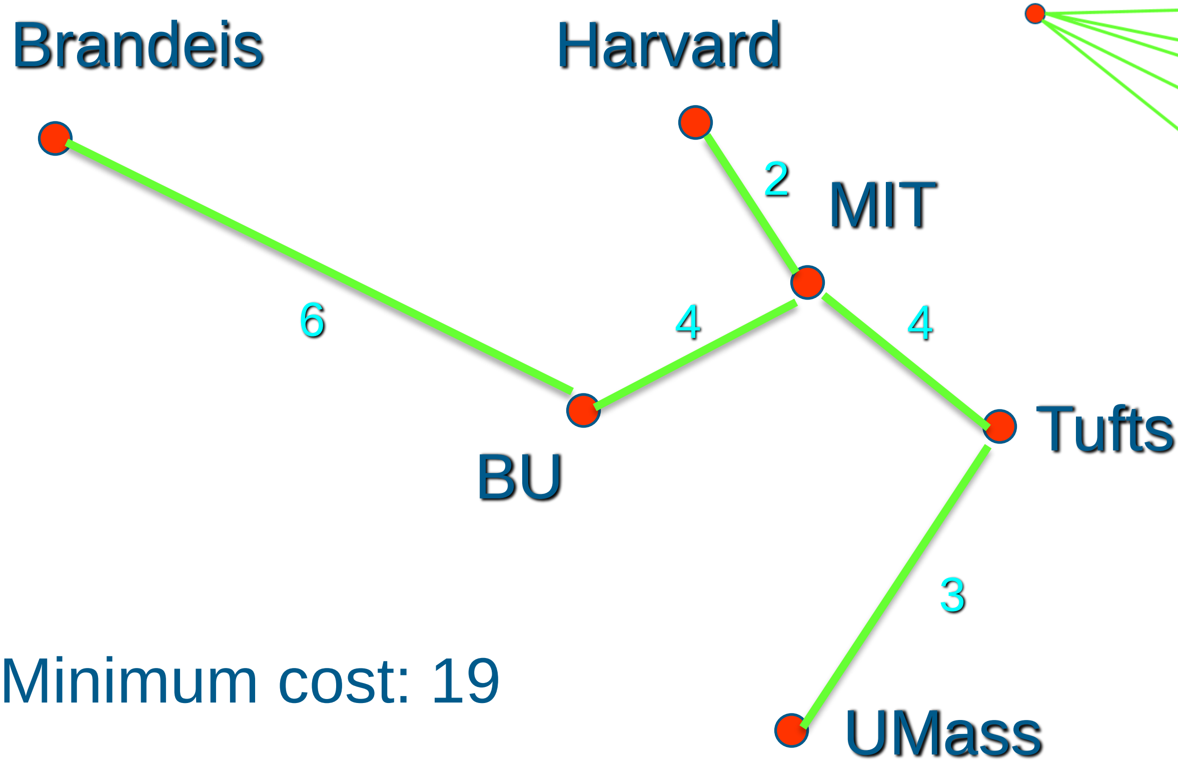
- Both algorithms are **guaranteed** to produce a minimum spanning tree of a connected weighted graph.

UMASS BOSTON

Spanning Trees

- **Example:** Use Kruskal's algorithm to design a minimum spanning tree for the libraries' graph.





Minimum cost: 19

Spanning Trees

- **Prim vs. Kruskal:**
- The two algorithms differ in the way they can be implemented and their efficiency under different conditions.

- As a rule of thumb, **Prim's algorithm** is more efficient when initially there are many **more edges** than vertices.

- For graphs with initially only **few edges** in comparison to the number of vertices, **Kruskal's algorithm** typically performs more efficiently.

- Boolean Algebra

Boolean Algebra

- Boolean algebra provides the operations and the rules for working with the set **{0, 1}.**
- These are the rules that underlie **electronic circuits**, and the methods we will discuss are fundamental to **VLSI design**.
- We are going to focus on three operations:
  - Boolean complementation,
  - Boolean sum, and
  - Boolean product

Boolean Operations

- The **complement** is denoted by a bar. It is defined by
- $\bar{0} = 1$ and $\bar{1} = 0$.
- The **Boolean sum**, denoted by + or by OR, has the following values:
- $1 + 1 = 1, \quad 1 + 0 = 1, \quad 0 + 1 = 1, \quad 0 + 0 = 0$
- The **Boolean product**, denoted by · or by AND, has the following values:
- $1 \cdot 1 = 1, \quad 1 \cdot 0 = 0, \quad 0 \cdot 1 = 0, \quad 0 \cdot 0 = 0$

UMASS BOSTON

Boolean Functions and Expressions

- **Definition:** Let $B = \{0, 1\}$. The variable x is called a **Boolean variable** if it assumes values only from B.

- A function from B^n, the set $\{(x_1, x_2, \ldots, x_n) \mid x_i \in B, 1 \leq i \leq n\}$, to B is called a **Boolean function of degree n**.

- Boolean functions can be represented using expressions made up from Boolean variables and Boolean operations.

Boolean Functions and Expressions

- **Question:** How many different Boolean functions of degree 1 are there?
- **Solution:** There are four of them, F_1, F_2, F_3, and F_4:

x	F_1	F_2	F_3	F_4
0	0	0	1	1
1	0	1	0	1

Boolean Functions and Expressions

- **Question:** How many different Boolean functions of degree 2 are there?
- **Solution:** There are 16 of them, F_1, F_2, …, F_{16}:

x	y	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Boolean Functions and Expressions

- **Question:** How many different Boolean functions of degree n are there?

- **Solution:**
- There are 2^n different n-tuples of 0s and 1s.
- A Boolean function is an assignment of 0 or 1 to each of these 2^n different n-tuples.
- Therefore, there are 2^{2^n} different Boolean functions

Boolean Functions and Expressions

- The **Boolean expressions** in the variables $x_1, x_2, \ldots, x_n$ are defined recursively as follows:
  - $0, 1, x_1, x_2, \ldots, x_n$ are Boolean expressions.
  - If E_1 and E_2 are Boolean expressions, then $(\overline{E_1})$, (E_1E_2), and $(E_1 + E_2)$ are Boolean expressions.

- Each Boolean expression represents a Boolean function. The values of this function are obtained by substituting 0 and 1 for the variables in the expression.

Boolean Functions and Expressions

- For example, we can create Boolean expression in the variables x, y, and z using the “building blocks” 0, 1, x, y, and z, and the construction rules:
- Since x and y are Boolean expressions, so is xy.
- Since z is a Boolean expression, so is $(\bar{z})$.
- Since xy and $(\bar{z})$ are Boolean expressions, so is xy + $(\bar{z})$.
- … and so on…

Boolean Functions and Expressions

- **Example:** Give a Boolean expression for the Boolean function F(x, y) as defined by the following table:

x	y	F(x, y)
0	0	0
0	1	1
1	0	0
1	1	0

Possible solution: F(x, y) = $(\overline{x}).y$

Boolean Functions and Expressions

- **Another Example:**

x	y	z	F(x, y, z)
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Possible solution I:

F(x, y, z) = $\overline{(xz + y)}$

Possible solution II:

F(x, y, z) = $\overline{(xz)}\,\overline{(y)}$

Boolean Functions and Expressions

- **Definition:** The Boolean functions F and G of n variables are **equal** if and only if $F(b_1, b_2, \ldots, b_n) = G(b_1, b_2, \ldots, b_n)$ whenever $b_1, b_2, \ldots, b_n$ belong to B.

- Two different Boolean expressions that represent the same function are called **equivalent**.

- For example, the Boolean expressions xy, xy + 0, and xy·1 are equivalent.

Boolean Functions and Expressions

- The **complement** of the Boolean function F is the function $\bar{F}$, where $\bar{F}(b_1, b_2, \ldots, b_n) = \overline{F(b_1, b_2, \ldots, bn)}$.

- Let F and G be Boolean functions of degree n. The **Boolean sum F+G** and **Boolean product FG** are then defined by

- $(F + G)(b_1, b_2, \ldots, b_n) = F(b_1, b_2, \ldots, b_n)+G(b_1, b_2, \ldots, b_n)$

- $(FG)(b_1, b_2, \ldots, b_n) = F(b_1, b_2, \ldots, b_n)\ G(b_1, b_2, \ldots, b_n)$

Identities

- There are useful identities of Boolean expressions that can help us to transform an expression A into an equivalent expression B, e.g.:

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0+x = x$
Null (or Dominance) Law	$0x = 0$	$1+x = 1$
Idempotent Law	$xx = x$	$x+x = x$
Inverse Law	$x\bar{x} = 0$	$x+\bar{x} = 1$
Commutative Law	$xy = yx$	$x+y = y+x$
Associative Law	$(xy)z = x(yz)$	$(x+y)+z = x+(y+z)$
Distributive Law	$x+yz = (x+y)(x+z)$	$x(y+z) = xy+xz$
Absorption Law	$x(x+y) = x$	$x+xy = x$
DeMorgan's Law	$(\overline{xy}) = \bar{x}+\bar{y}$	$(\overline{x+y}) = \bar{x}\bar{y}$
Double Complement Law	$\bar{\bar{x}} = x$	

Identities

- These identities come in pairs. To explain the relationship between the two identities in each pair we use the concept of a dual.
- The **dual** of a Boolean expression is obtained by interchanging Boolean sums and Boolean products and interchanging 0s and 1s.
- The dual of a Boolean function F represented by a Boolean expression is the function represented by the dual of this expression.
- This dual function, denoted by F^d, does not depend on the particular Boolean expression used to represent F.
- An identity between functions represented by Boolean expressions remains valid when the duals of both sides of the identity are taken. This result, called the **duality principle**, is useful for obtaining new identities.

Definition of a Boolean Algebra

- All the properties of Boolean functions and expressions that we have discovered also apply to **other mathematical structures** such as propositions and sets and the operations defined on them.
- If we can show that a particular structure is a Boolean algebra, then we know that all results established about Boolean algebras apply to this structure.
- For this purpose, we need an **abstract definition** of a Boolean algebra.

Definition of a Boolean Algebra

- **Definition:** A Boolean algebra is a set B with two binary operations $\vee$ and $\wedge$, elements 0 and 1, and a unary operation $\bar{\ }$ such that the following properties hold for all x, y, and z in B:
- $x \vee 0 = x$ and $x \wedge 1 = x$ (identity laws)
- $x \vee (\bar{x}) = 1$ and $x \wedge (\bar{x}) = 0$ (domination laws)
- $(x \vee y) \vee z = x \vee (y \vee z)$ and
$(x \wedge y) \wedge z = x \wedge (y \wedge z)$ (associative laws)
- $x \vee y = y \vee x$ and $x \wedge y = y \wedge x$ (commutative laws)
- $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ and
$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ (distributive laws)

Representing Boolean Functions

- There is a simple method for deriving a Boolean expression for a function that is defined by a table. This method is based on **minterms**.

- **Definition:** A **literal** is a Boolean variable or its complement. A **minterm** of the Boolean variables x_1, x_2, …, x_n is a Boolean product $y_1y_2\ldots y_n$, where $y_i = x_i$ or $y_i = \overline{x_i}$.

- Hence, a minterm is a product of n literals, with one literal for each variable.

Representing Boolean Functions

- Consider F(x,y,z) again:

x	y	z	F(x, y, z)
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

F(x, y, z) = 1 if and only if:

x = y = z = 0 or

x = y = 0, z = 1 or

x = 1, y = z = 0

Therefore,

$F(x, y, z) = (\overline{x})(\overline{y})(\overline{z}) + (\overline{x})(\overline{y})z + x(\overline{y})(\overline{z})$

Logic Gates

- Electronic circuits consist of so-called gates. There are three basic types of gates: