

Chapter 5

Trees

By:Siti Hazyanti

Introduction to Trees

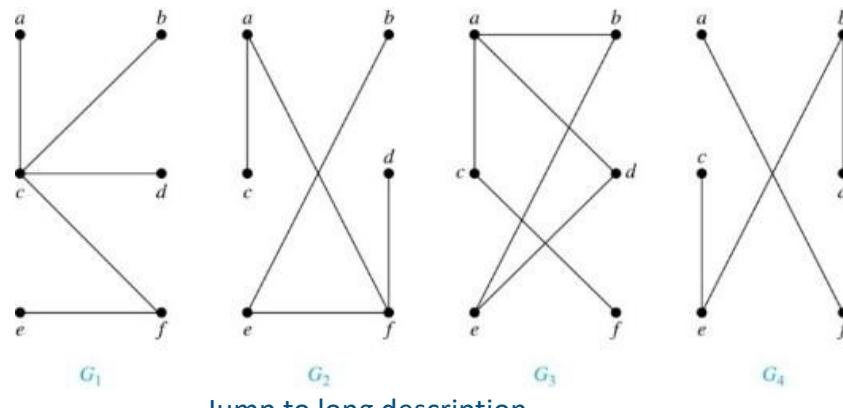
Section 5.1

Trees₁

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

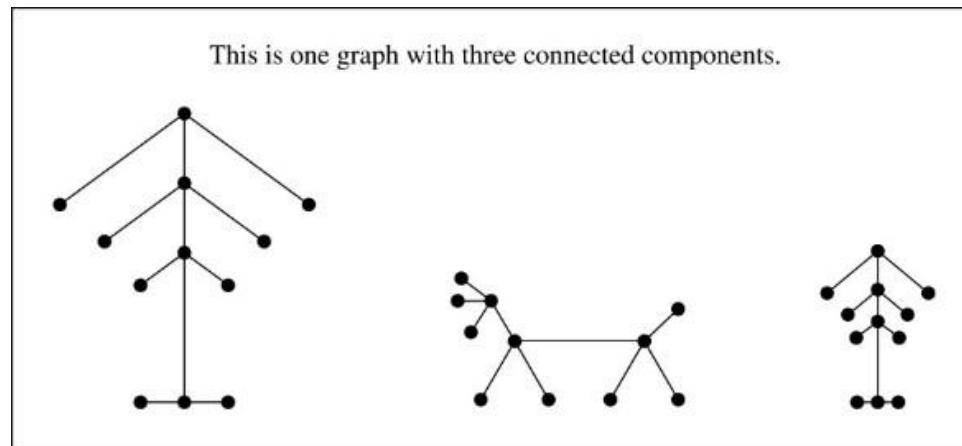
Example: Which of these graphs are trees?

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

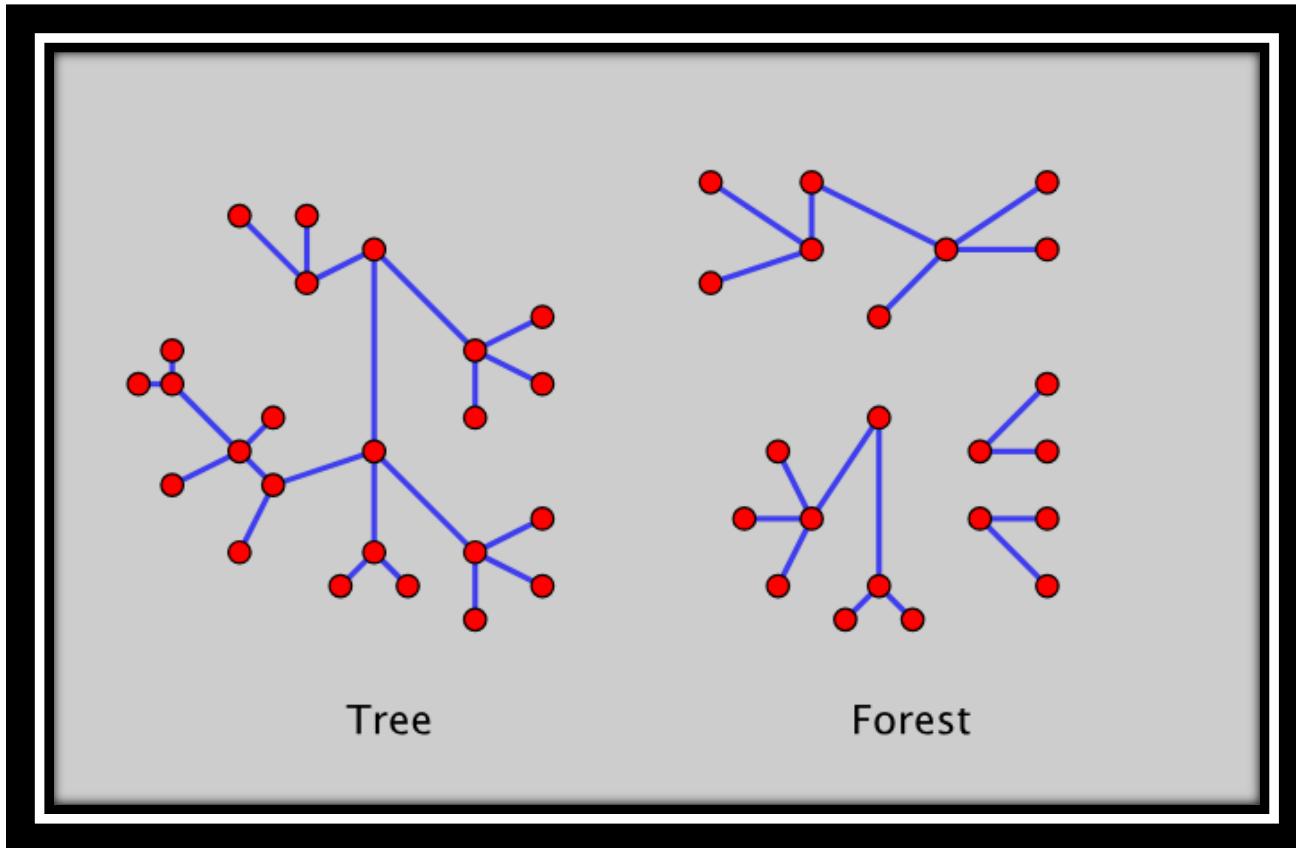


[Jump to long description](#)

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



Forest vs Tree



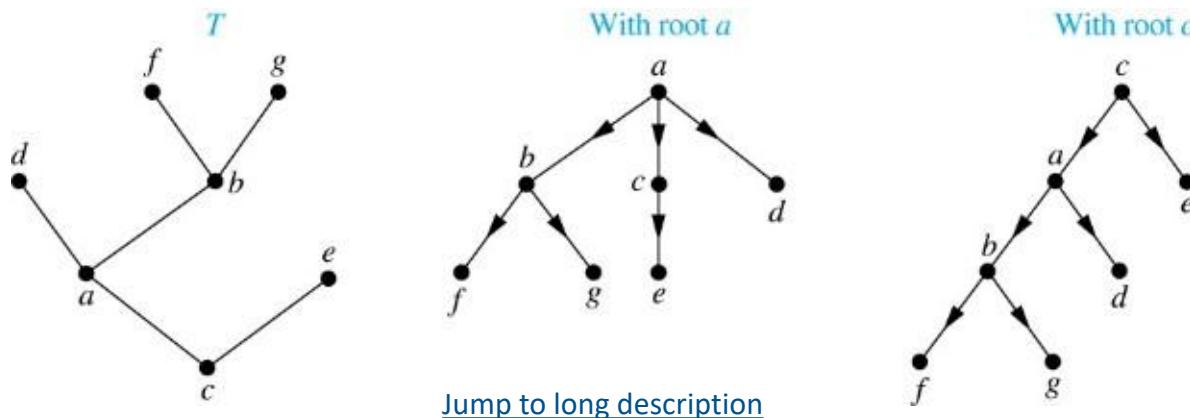
Tree

Forest

Rooted Trees

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

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



Terminology for Rooted Trees

Parent- parent of h is g

Children- children of h is k

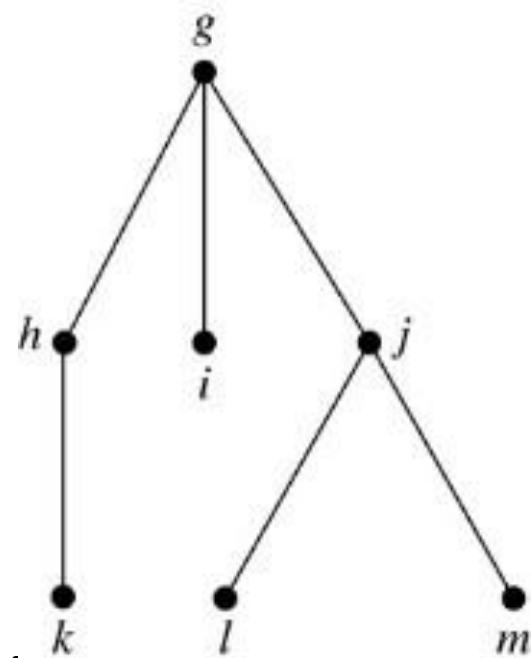
Siblings- sibling of h is i and j

Ancestors- ancestors of h is g

Descendants- descendant of h is k

Leaves- all leaves k,i,l,m

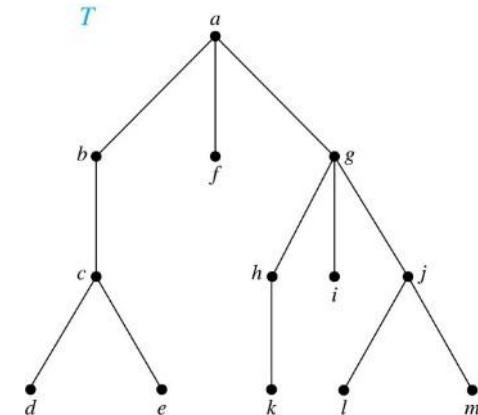
internal vertices- h,g,j



Terminology for Rooted Trees

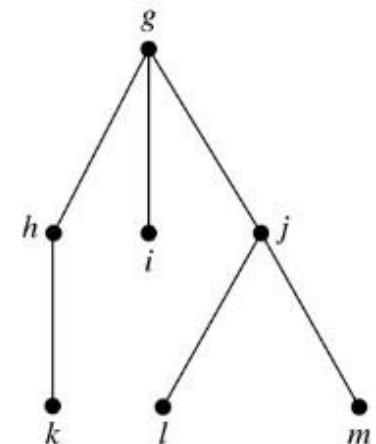
Example: In the rooted tree T (with root a):

- Find the parent of c , the children of g , the siblings of h , the ancestors of e , and the descendants of b .
- Find all internal vertices and all leaves.
- What is the subtree rooted at G ?



Solution:

- The parent of c is b . The children of g are h , i , and j .
The siblings of h are i and j . The ancestors of e are c , b , and a . The descendants of b are c , d , and e .
- The internal vertices are a , b , c , g , h , and j . The leaves are d , e , f , i , k , l , and m .
- We display the subtree rooted at g .



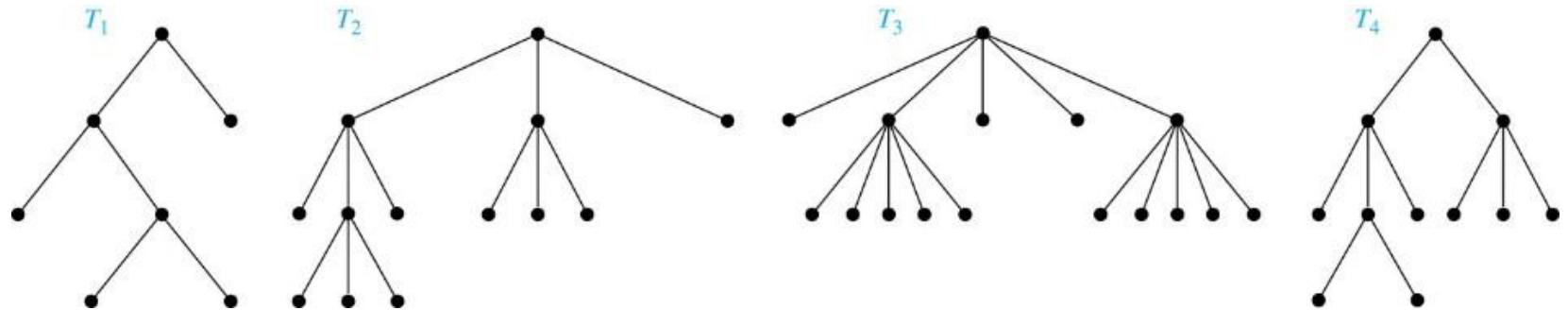
[Jump to long description](#)

m-ary Rooted Trees

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

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

[Jump to long description](#)



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

Ordered Rooted Trees

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

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

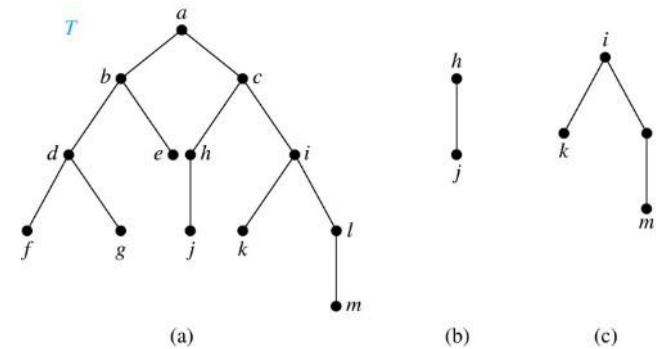
Definition: A *binary tree* is an ordered rooted where each internal vertex has at most two children. If an internal vertex of a binary tree has two children, the first is called the *left child* and the second the *right child*. The tree rooted at the left child of a vertex is called the *left subtree*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

Example: Consider the binary tree T .

- I. What are the left and right children of d ?
- II. What are the left and right subtrees of c ?

Solution:

- I. The left child of d is f and the right child is g .
- II. The left and right subtrees of c are displayed in (b) and (c).



[Jump to long description](#)

Level of vertices and height of trees

When working with trees, we often want to have rooted trees where the subtrees at each vertex contain paths of approximately the same length.

To make this idea precise we need some definitions:

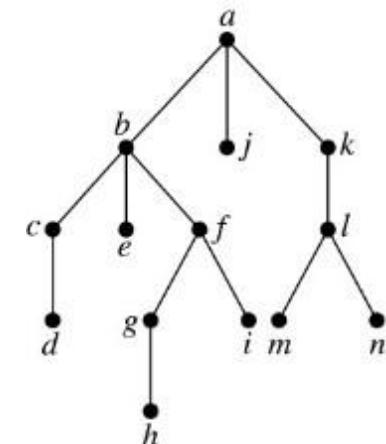
- The *level* of a vertex v in a rooted tree is the length of the unique path from the root to this vertex.
- The *height* of a rooted tree is the maximum of the levels of the vertices.

Example:

- I. Find the level of each vertex in the tree to the right.
- II. What is the height of the tree?

Solution:

- I. The root a is at level 0. Vertices b, j , and k are at level 1. Vertices c, e, f , and l are at level 2. Vertices d, g, i, m , and n are at level 3. Vertex h is at level 4.
- II. The height is 4, since 4 is the largest level of any vertex.

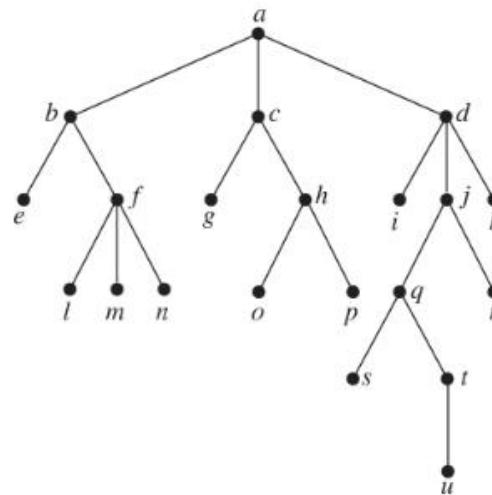


[Jump to long description](#)

Exercise 1

Answer the questions about the rooted tree illustrated.

- a) Which vertex is the root?
- b) Which vertices are internal?
- c) Which vertices are leaves?
- d) Which vertices are children of f ?
- e) Which vertex is the parent of h ?
- f) Which vertices are siblings of o ?
- g) Which vertices are ancestors of m ?
- h) Which vertices are descendants of b ?



Tree Traversal

Section 5.2

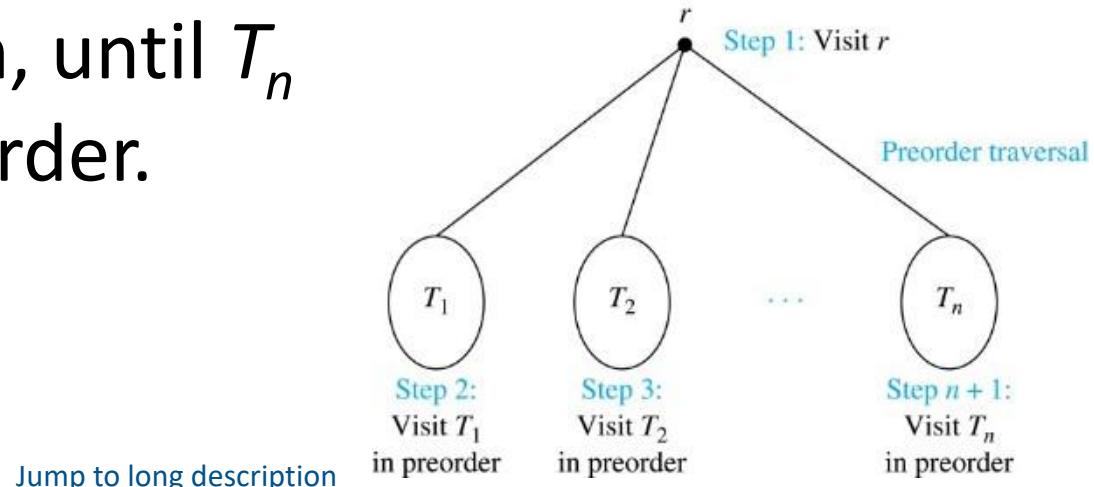
Tree Traversal

Procedures for systematically visiting every vertex of an ordered tree are called *traversals*.

The three most commonly used *traversals* are *preorder traversal*, *inorder traversal*, and *postorder traversal*.

Preorder Traversal₁

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



Preorder Traversal₂

procedure *preorder* (T : ordered rooted tree)

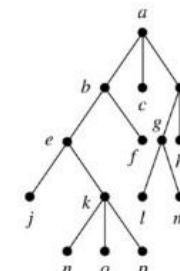
$r :=$ root of T

list r

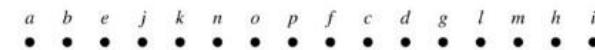
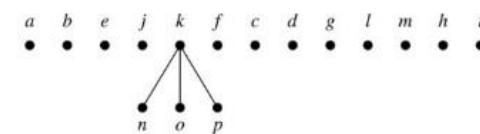
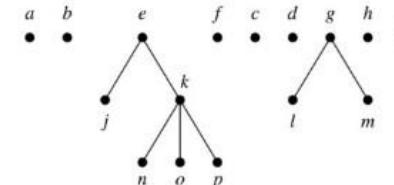
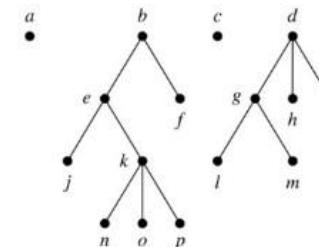
for each child c of r from left to right

$T(c) :=$ subtree with c as root

preorder($T(c)$)



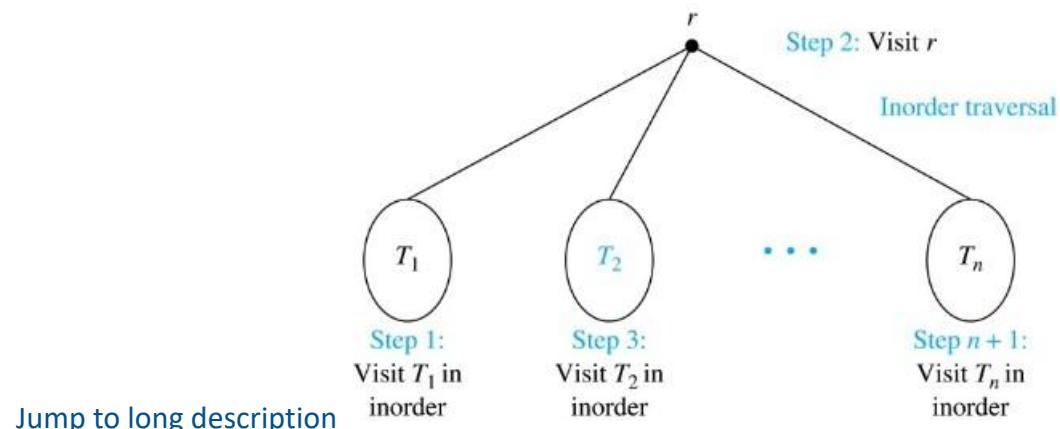
Preorder traversal: Visit root, visit subtrees left to right



[Jump to long description](#)

Inorder Traversal₁

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



Inorder Traversal₂

procedure *inorder* (*T*: ordered rooted tree)

r := root of *T*

if *r* is a leaf **then** list *r*

else

l := first child of *r* from left to right

T(l) := subtree with *l* as its root

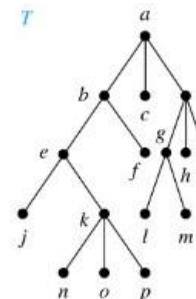
inorder(*T(l)*)

list(*r*)

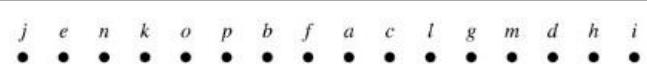
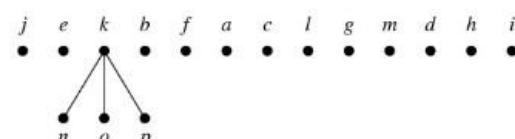
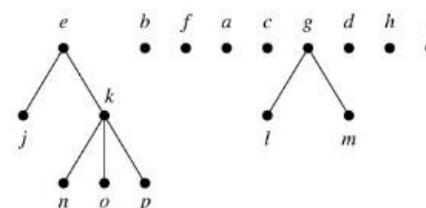
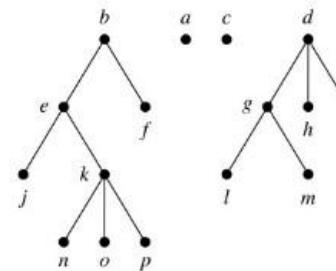
for each child *c* of *r* from left to right

T(c) := subtree with *c* as root

inorder(*T(c)*)



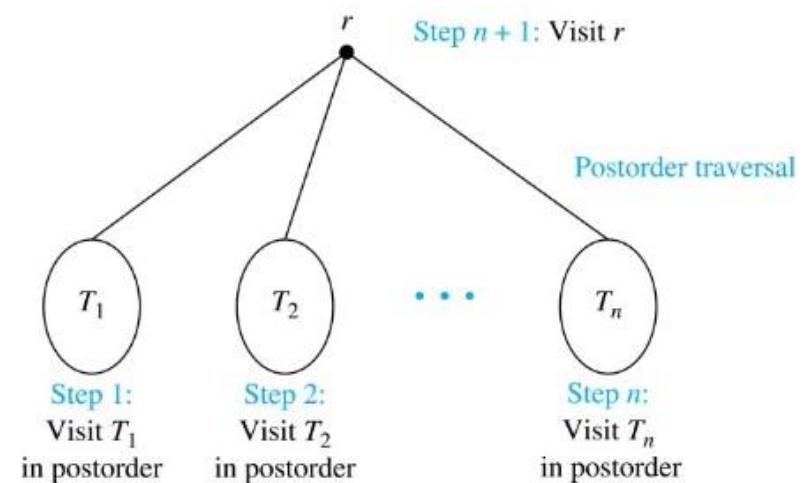
Inorder traversal: Visit leftmost subtree, visit root, visit other subtrees left to right



[Jump to long description](#)

Postorder Traversal₁

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



[Jump to long description](#)

Postorder Traversal₂

procedure *postorder* (T : ordered rooted tree)

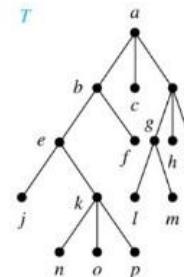
$r :=$ root of T

for each child c of r from left to right

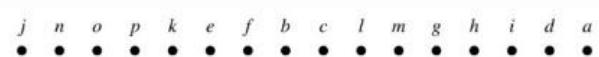
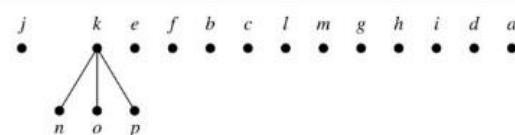
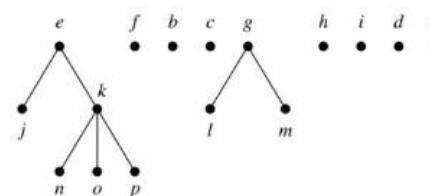
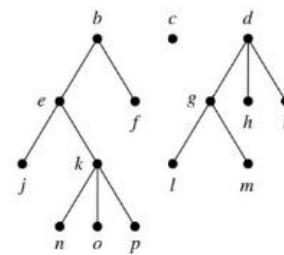
$T(c) :=$ subtree with c as root

$\text{postorder}(T(c))$

list r



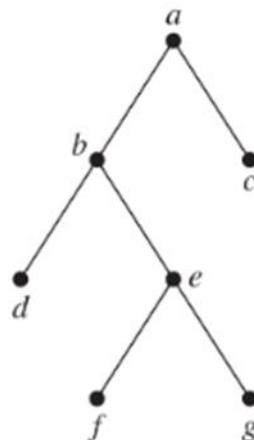
Postorder traversal: Visit subtrees left to right; visit root



[Jump to long description](#)

Exercise 2

Determine the order in which a preorder traversal visits the vertices of the given ordered rooted tree.



Spanning Trees

Section 9.3

Section Summary₃

Spanning Trees

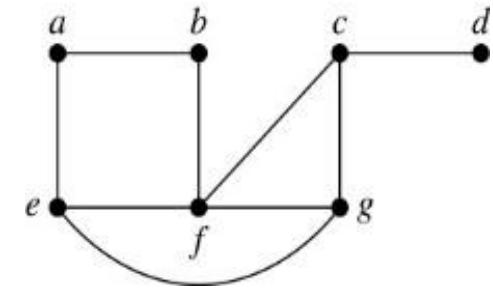
Depth-First Search

Breadth-First Search

Depth-First Search in Directed Graphs

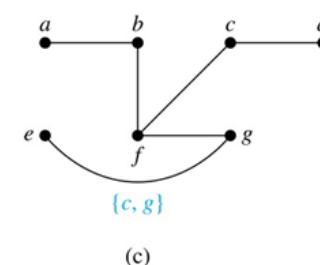
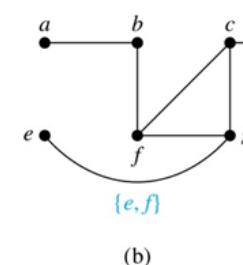
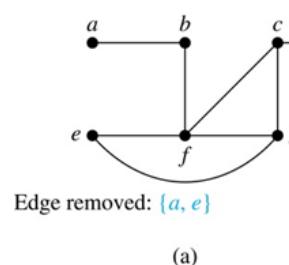
Spanning Trees₁

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



Example: Find the spanning tree of this simple graph:

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



(a)

(b)

(c)

Spanning Trees₂

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

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

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

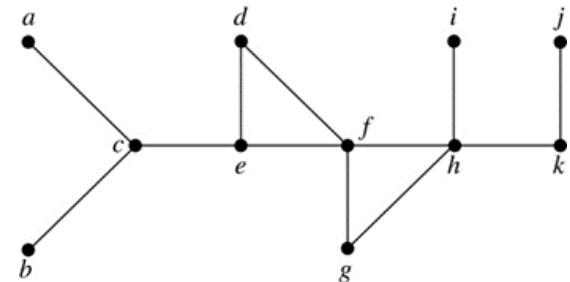
Depth-First Search₁

To use *depth-first search* to build a spanning tree for a connected simple graph first arbitrarily choose a vertex of the graph as the root.

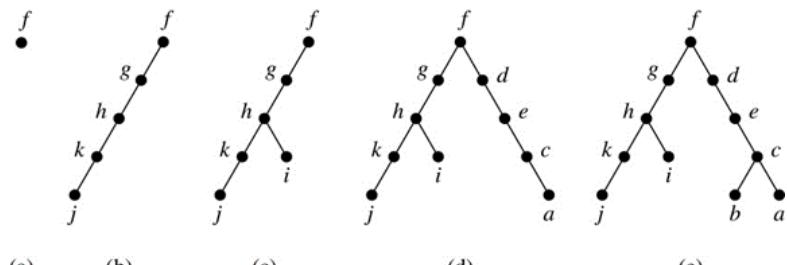
- Form a path starting at this vertex by successively adding vertices and edges, where each new edge is incident with the last vertex in the path and a vertex not already in the path. Continue adding vertices and edges to this path as long as possible.
- If the path goes through all vertices of the graph,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

Depth-First Search₂

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



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

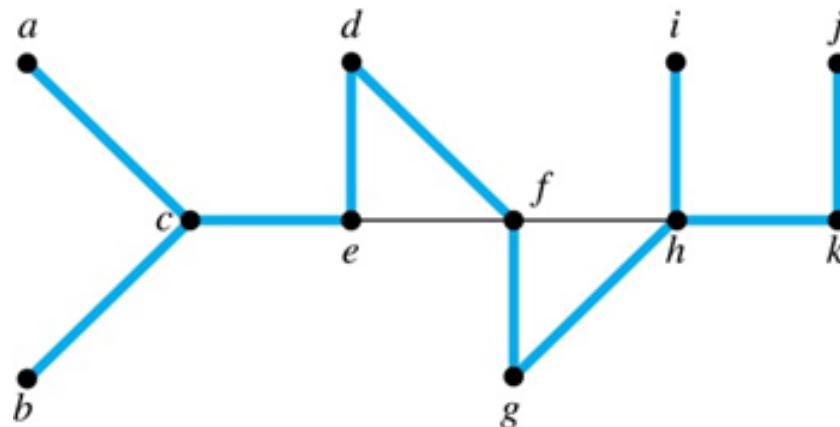


[Jump to long description](#)

Depth-First Search₃

The edges selected by depth-first search of a graph are called *tree edges*. All other edges of the graph must connect a vertex to an ancestor or descendant of the vertex in the graph. These are called *back edges*.

In this figure, the tree edges are shown with heavy blue lines. The two thin black edges are back edges.



Depth-First Search Algorithm

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

```
procedure DFS( $G$ : connected graph with vertices  $v_1, v_2, \dots, v_n$ )  
     $T :=$  tree consisting only of the vertex  $v_1$   
    visit( $v_1$ )  
  
procedure visit( $v$ : vertex of  $G$ )  
    for each vertex  $w$  adjacent to  $v$  and not yet in  $T$   
        add vertex  $w$  and edge  $\{v,w\}$  to  $T$   
        visit( $w$ )
```

Breadth-First Search₁

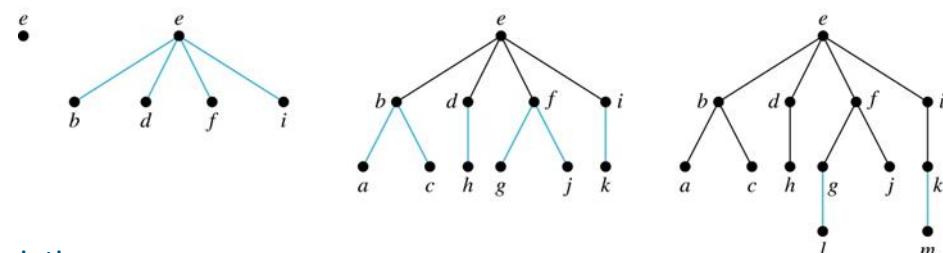
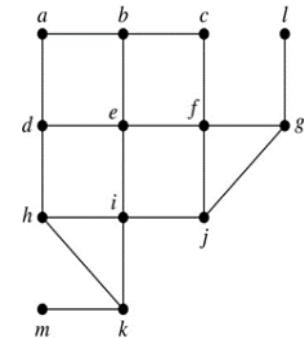
We can construct a spanning tree using *breadth-first search*. We first arbitrarily choose a root from the vertices of the graph.

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

Breadth-First Search₂

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

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



[Jump to long description](#)

Breadth-First Search Algorithm

We now use pseudocode to describe breadth-first search.

procedure $BFS(G:$ connected graph with vertices $v_1, v_2, \dots, v_n)$

$T :=$ tree consisting only of the vertex v_1

$L :=$ empty list $visit(v_1)$

put v_1 in the list L of unprocessed vertices

while L is not empty

remove the first vertex, v , from L

for each neighbor w of v

if w is not in L and not in T **then**

add w to the end of the list L

add w and edge $\{v,w\}$ to T

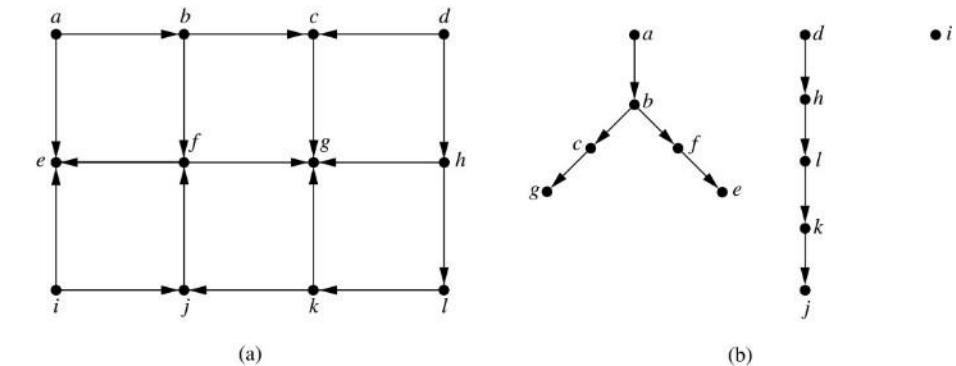
Depth-First Search in Directed Graphs

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

Example: For the graph in (a), if we begin at vertex a , depth-first search adds the path connecting a, b, c , and g . At g , we are blocked, so we return to c . Next, we add the path connecting f to e . Next, we return to a

and find that we cannot add a new path. So, we begin another tree with d as its root. We find that this new tree consists of the path connecting the vertices d, h, l, k , and j . Finally, we add a new tree, which only contains i , its root.

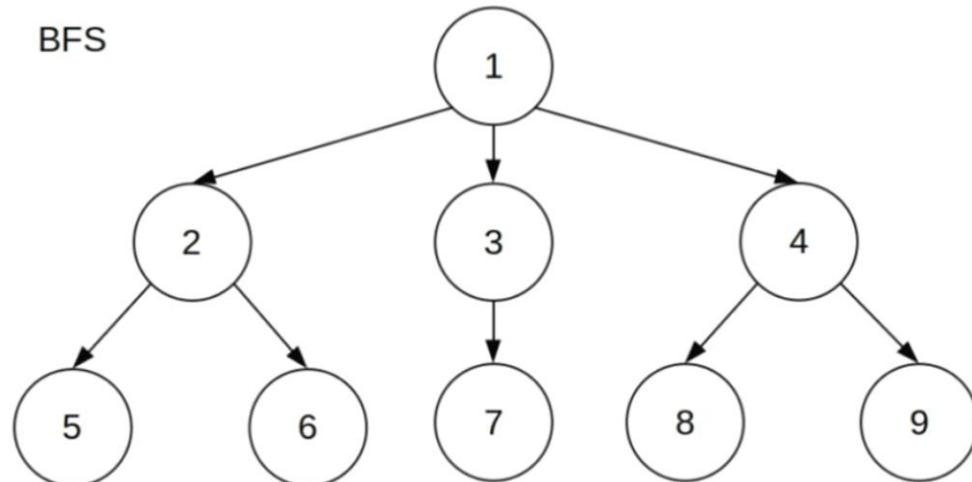
To index websites, search engines such as Google systematically explore the web starting at known sites. The programs that do this exploration are known as *Web spiders*. They may use both breath-first search or depth-first search to explore the Web graph.



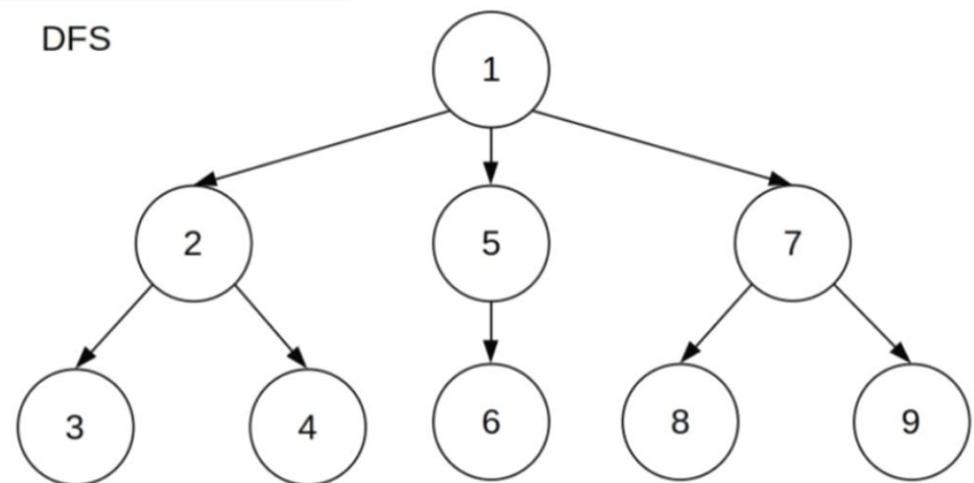
[Jump to long description](#)

Comparison

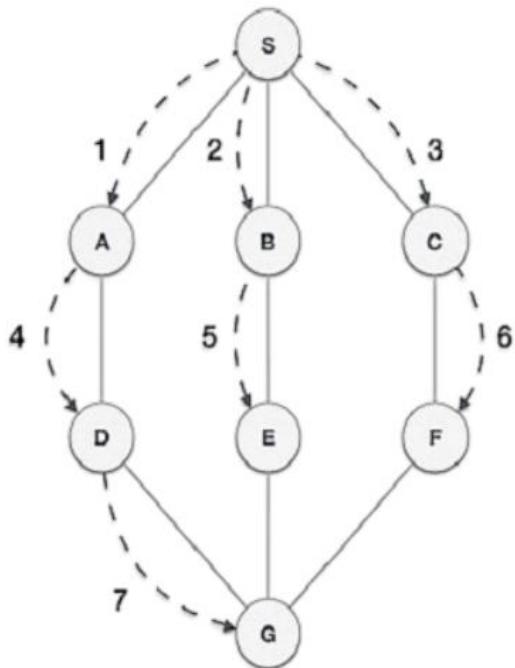
BFS



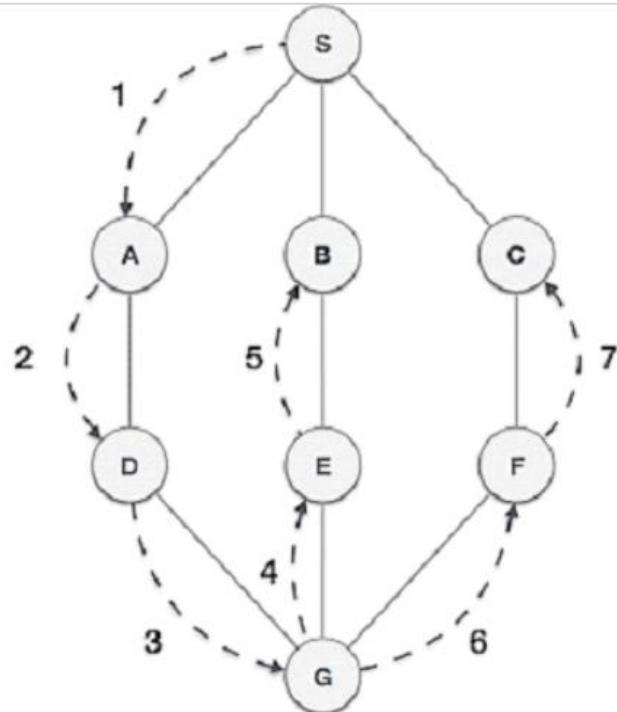
DFS



Example of BFS



Example of DFS



Extra notes

Differences between BFS and DFS

The following are the differences between the BFS and DFS:

	BFS	DFS
Full form	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
Technique	It a vertex-based technique to find the shortest path in a graph.	It is an edge-based technique because the vertices along the edge are explored first from the starting to the end node.
Definition	BFS is a traversal technique in which all the nodes of the same level are explored first, and then we move to the next level.	DFS is also a traversal technique in which traversal is started from the root node and explore the nodes as far as possible until we reach the node that has no unvisited adjacent nodes.

Data Structure	Queue data structure is used for the BFS traversal.	Stack data structure is used for the BFS traversal.
Backtracking	BFS does not use the backtracking concept.	DFS uses backtracking to traverse all the unvisited nodes.
Number of edges	BFS finds the shortest path having a minimum number of edges to traverse from the source to the destination vertex.	In DFS, a greater number of edges are required to traverse from the source vertex to the destination vertex.
Optimality	BFS traversal is optimal for those vertices which are to be searched closer to the source vertex.	DFS traversal is optimal for those graphs in which solutions are away from the source vertex.
Speed	BFS is slower than DFS.	DFS is faster than BFS.
Suitability for decision tree	It is not suitable for the decision tree because it requires exploring all the neighboring nodes first.	It is suitable for the decision tree. Based on the decision, it explores all the paths. When the goal is found, it stops its traversal.
Memory efficient	It is not memory efficient as it requires more memory than DFS.	It is memory efficient as it requires less memory than BFS.