Trees

Chapter 11

With Question/Answer Animations

Chapter Summary

Introduction to Trees

Applications of Trees (*not currently included in overheads*)

Tree Traversal

Spanning Trees

Minimum Spanning Trees (*not currently included in overheads*)

Introduction to Trees

Section 11.1

Section Summary 1

Introduction to Trees

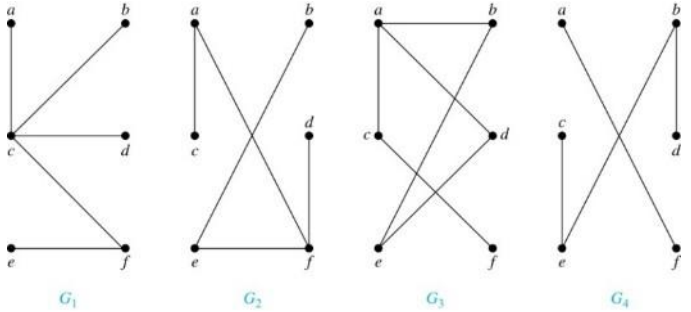
Rooted Trees

Trees as Models

Properties of Trees

Trees 1

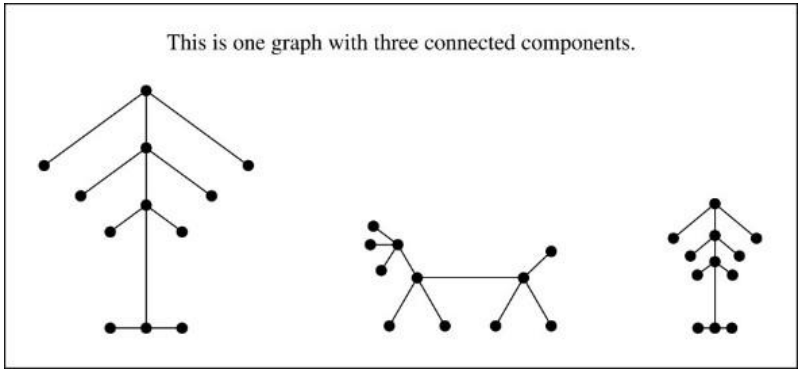
Definition: A *tree* is a connected undirected graph with no simple circuits.



Example: Which of these graphs are trees?

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

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



Jump to long description

Trees 2

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

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

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

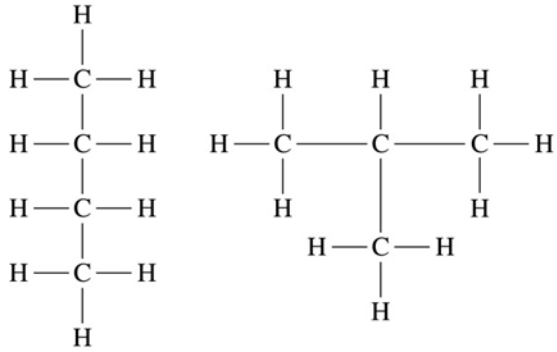
Hence, a graph with a unique simple path between any two vertices is a tree.

Trees as Models

Arthur Cayley
(1821-1895)

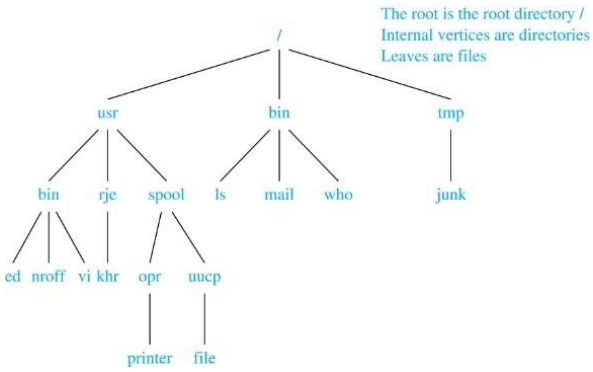
Trees are used as models in computer science, chemistry, geology, botany, psychology, and many other areas.

Trees were introduced by the mathematician Cayley in 1857 in his work counting the number of isomers of saturated hydrocarbons. The two isomers of butane are shown at the right.

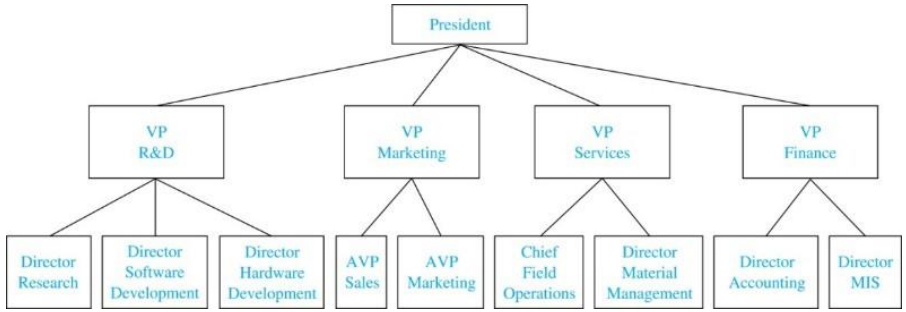


Butane Isobutane

The organization of a computer file system into directories, subdirectories, and files is naturally represented as a tree.



Trees are used to represent the structure of organizations.

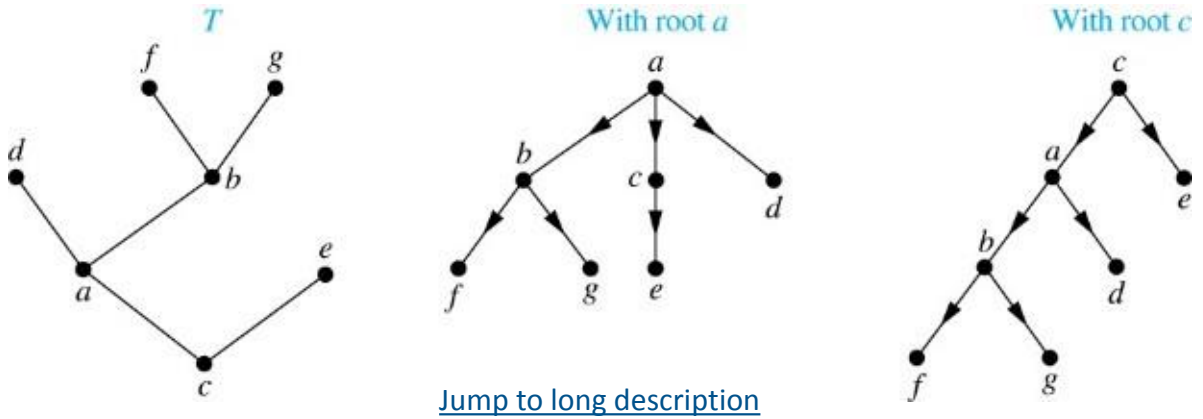


Jump to long description

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.



Jump to long description

Rooted Tree Terminology

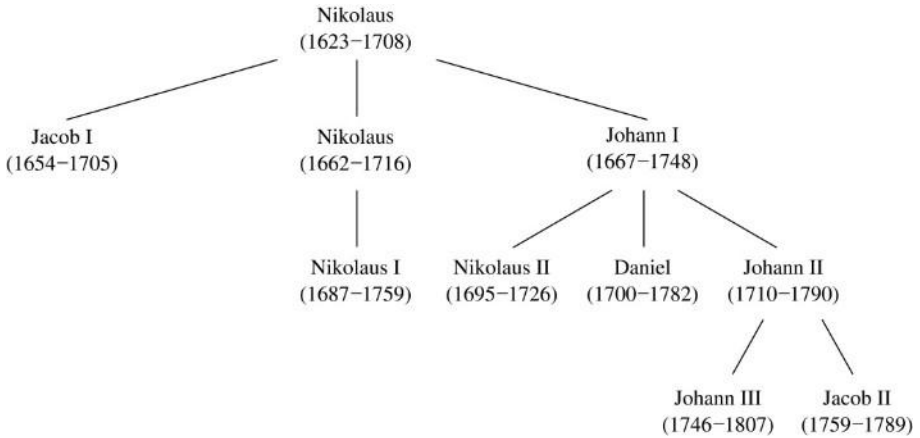
Terminology for rooted trees is a mix from botany and genealogy (such as this family tree of the Bernoulli family of mathematicians).

If *v* is a vertex of a rooted tree other than the root, the *parent* of *v* is the unique vertex *u* such that there is a directed edge from *u* to *v*. When *u* is a parent of *v*, *v* is called a *child* of *u*. Vertices with the same parent are called *siblings*.

The *ancestors* of a vertex are the vertices in the path from the root to this vertex, including the vertex itself and including the root. The *descendants* of a vertex *v* are those vertices that have *v* as an ancestor. (NB: proper ancestor/descendant)

A vertex of a rooted tree with no children is called a *leaf*. Vertices that have children are called *internal vertices*.

If *a* is a vertex in a tree, the *subtree* with *a* as its root is the subgraph of the tree consisting of *a* and its descendants and all edges incident to these descendants.



Jump to long description

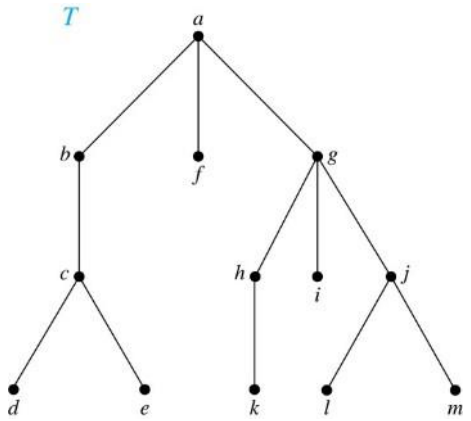
Terminology for Rooted Trees

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

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

(ii) Find all internal vertices and all leaves.

(iii) What is the subtree rooted at *g*?

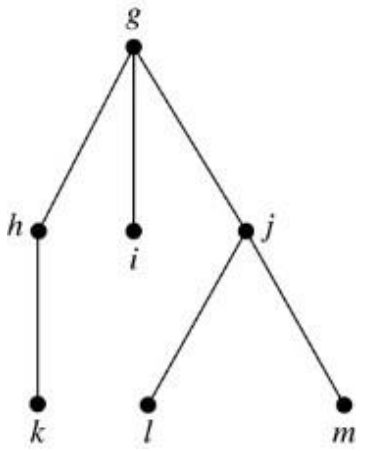


Solution:

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

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

(iii) We display the subtree rooted at g.

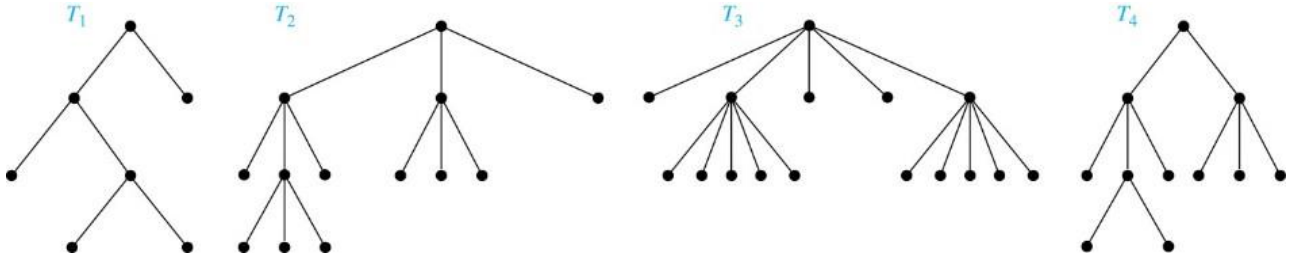


Jump to long description

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*?



Solution: T1 is a full binary tree because each of its internal vertices has two children. T2 is a full 3-ary tree because each of its internal vertices has three children. In T3 each internal vertex has five children, so T3 is a full 5-ary tree. T4 is not a full m-ary tree for any m because some of its internal vertices have two children and others have three children.

Jump to long description

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.

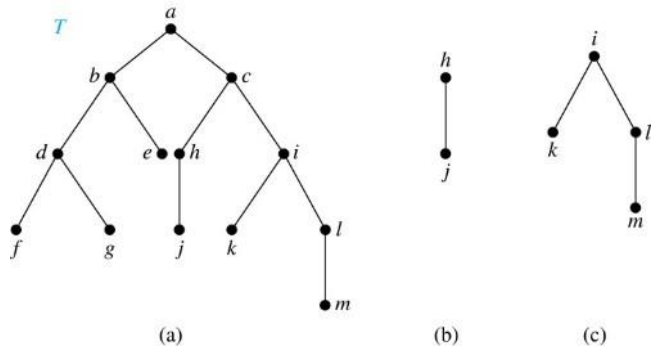
Definition: A *binary tree* is an ordered rooted where each internal vertex has at most two children. If an internal vertex of a binary tree has two children, the first is called the *left child* and the second the *right child*. The tree rooted at the left child of a vertex is called the *left subtree*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

Properties of Trees

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

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

BASIS STEP: LHS: When $n = 1$, a tree with one vertex has 0 edges. RHS: $n - 1 = 1 - 1 = 0$ ✓

INDUCTIVE HYPOTHESIS: Assume that every tree with k vertices has $k - 1$ edges for some k.

INDUCTIVE STEP: Show: a tree with $k + 1$ vertices has k edges Suppose a tree T has $k + 1$ vertices and that v is a leaf of T. Let w be the parent of v. Removing the vertex v and the edge connecting w to v produces a tree T' with k vertices. By the inductive hypothesis, T' has $k - 1$ edges. Because T has one more edge than T', we see that T has k edges.

Counting Vertices in Full *m*-Ary Trees 1

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

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

Counting Vertices in Full *m*-Ary Trees 2

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

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

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

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

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

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

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

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.

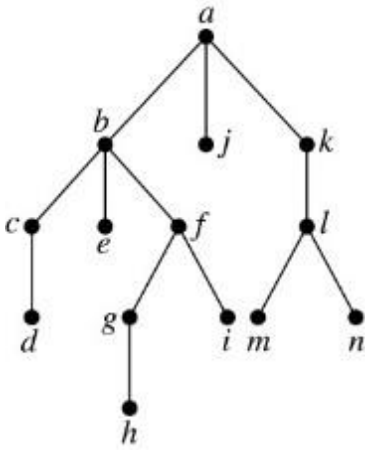
To make this idea precise we need some definitions:

- 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.

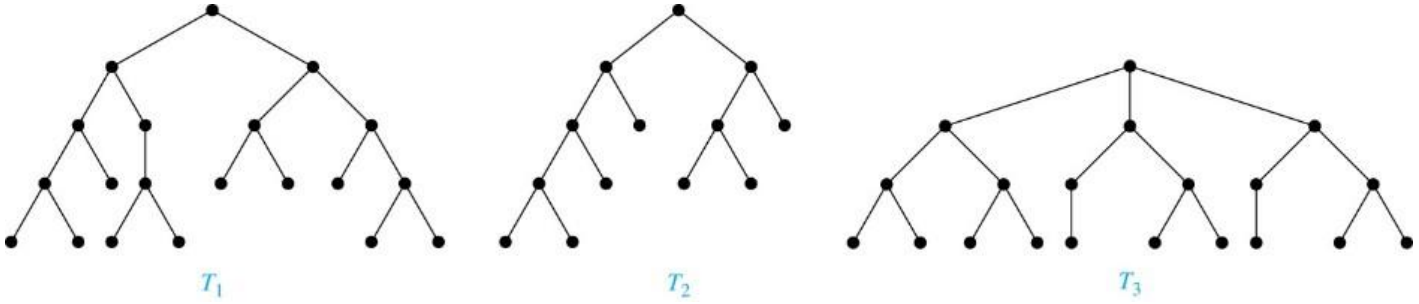
II. The height is 4, since 4 is the largest level of any vertex.

Jump to long description

Balanced *m*-Ary Trees

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

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



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

Jump to long description

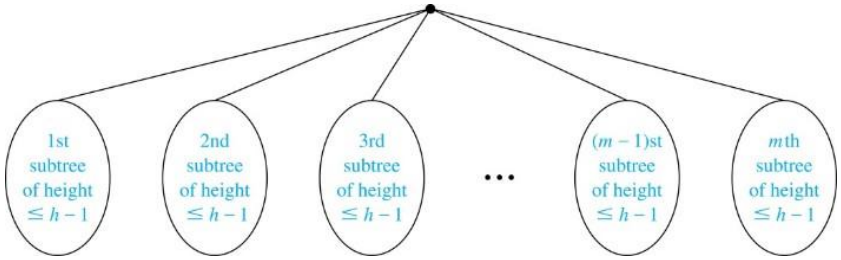
The Bound for the Number of Leaves in an *m*-Ary Tree

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

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

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

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



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

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

Tree Traversal

Section 11.3

Section Summary 2

Universal Address Systems (*not currently included in overheads*)

Traversal Algorithms

Infix, Prefix, and Postfix Notation

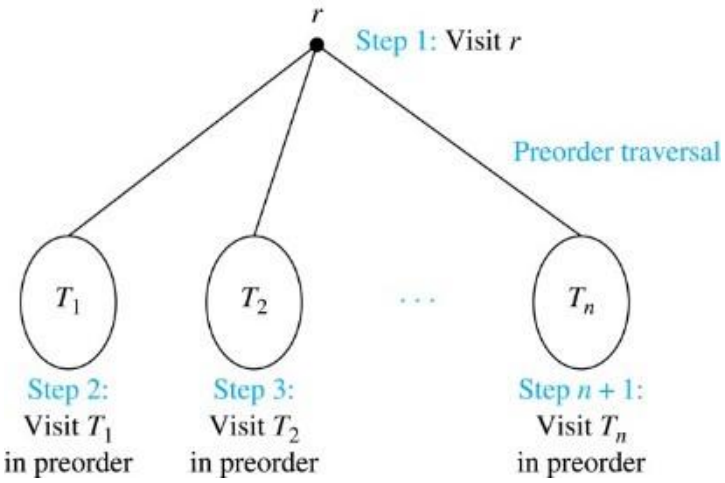
Tree Traversal

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 Traversal[1]

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



Jump to long description

Preorder Traversal 2

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

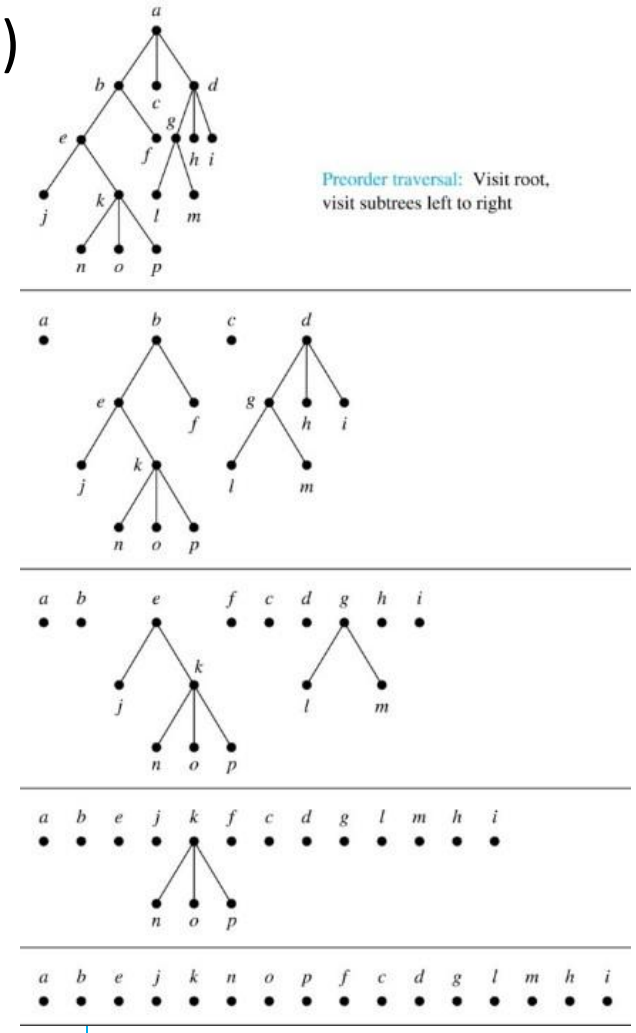
r := root of *T*

list *r*

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

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

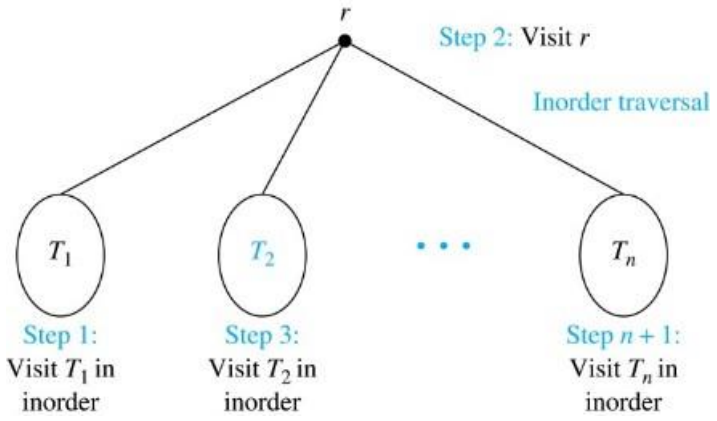
preorder(*T*(*c*))



Jump to long description

Inorder Traversal 1

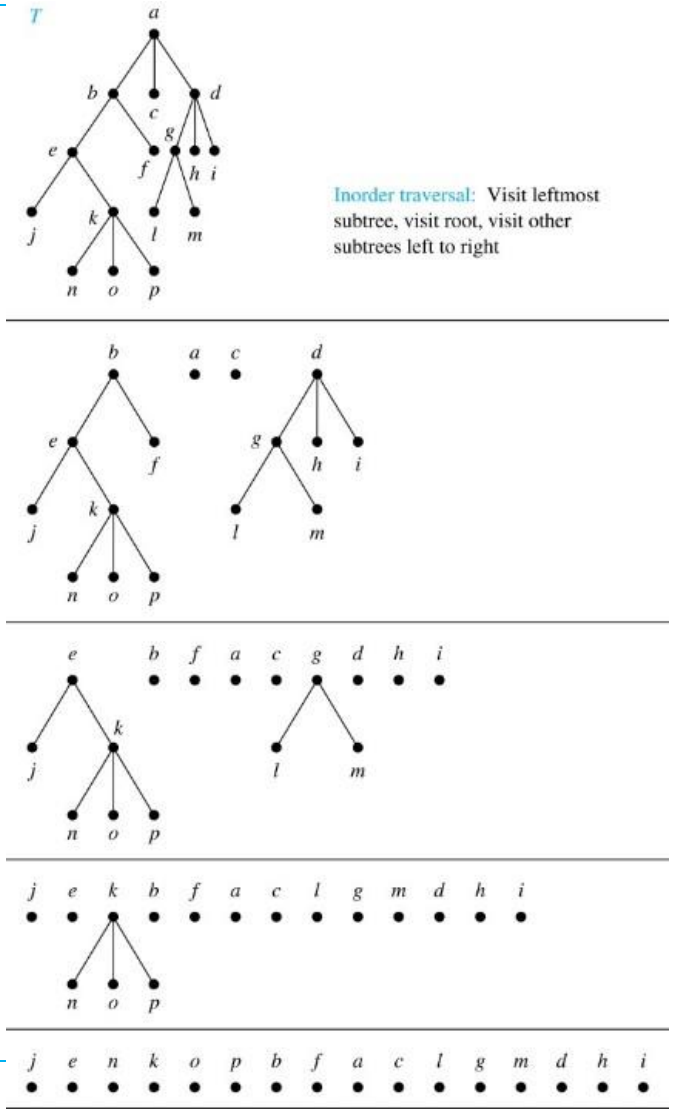
Definition: Let T be an ordered rooted tree with root r. If T consists only of r, then r is the *inorder traversal* of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees of r from left to right in T. The inorder traversal begins by traversing T_1 in inorder, then visiting r, and continues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



Jump to long description

Inorder Traversal 2

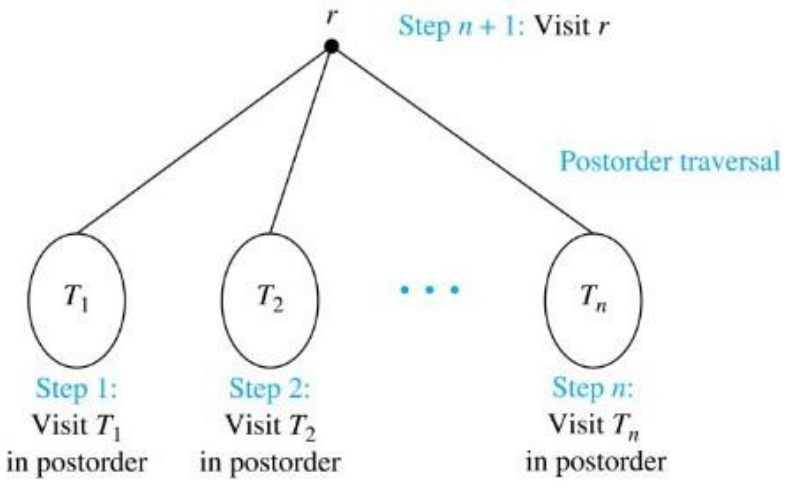
```
procedure inorder (T: ordered rooted tree)
    r := root of T
    if r is a leaf then
        list r
    else
        l := first child of r from left to right
        T(l) := subtree with l as its root
        inorder(T(l))
        list(r)
        for each child c of r from left to right
       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



Jump to long description

Postorder Traversal 1

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 $T_1, T_2, \ldots, T_n$ 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 Traversal 2

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

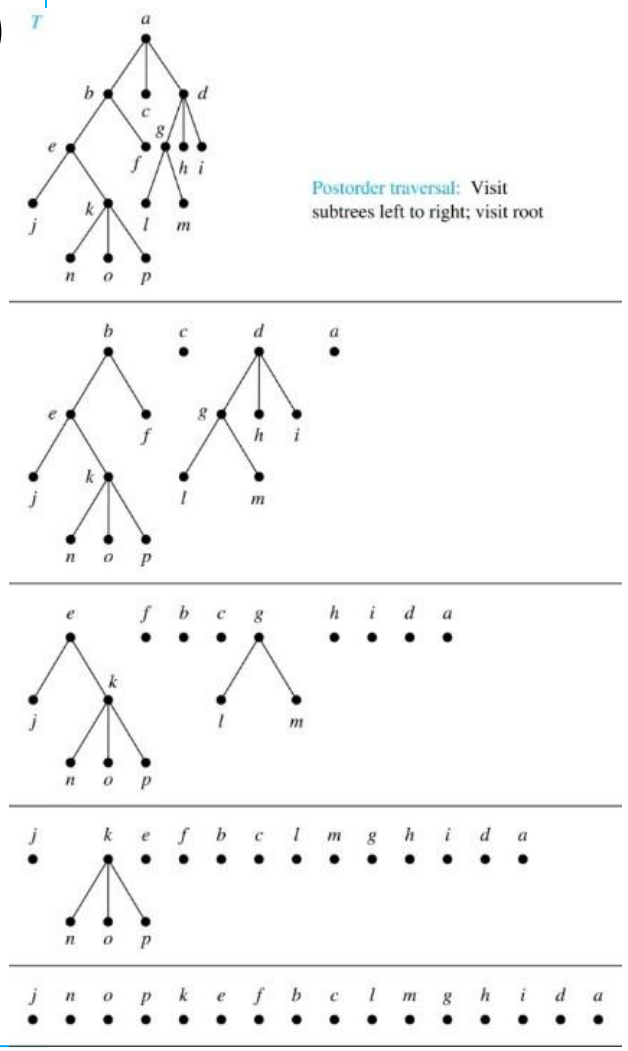
r := root of *T*

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

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

postorder(*T*(*c*))

list *r*



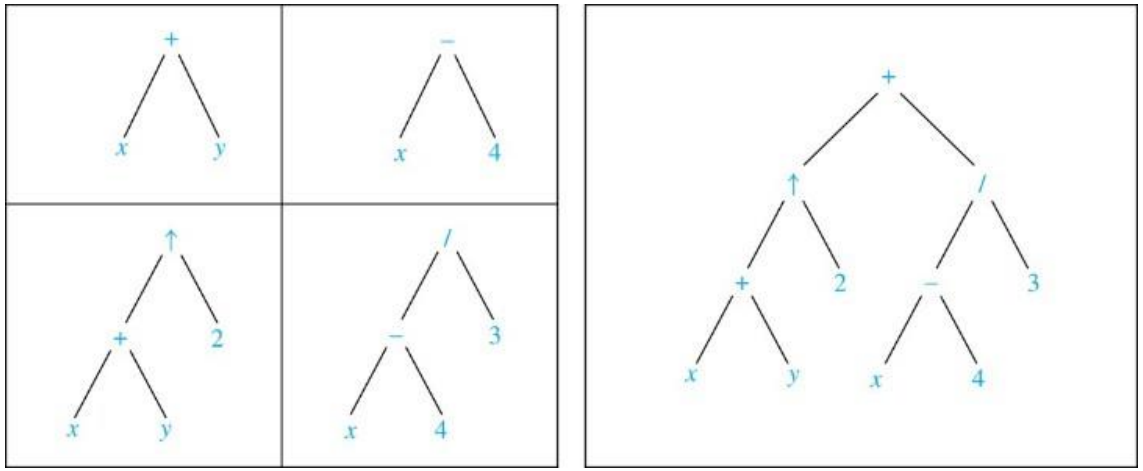
Jump to long description

Expression Trees

Complex expressions can be represented using ordered rooted trees.

Consider the expression $\left((x+y)\uparrow 2\right)+\left((x-4)/3\right)$.

A binary tree for the expression can be built from the bottom up, as is illustrated here.

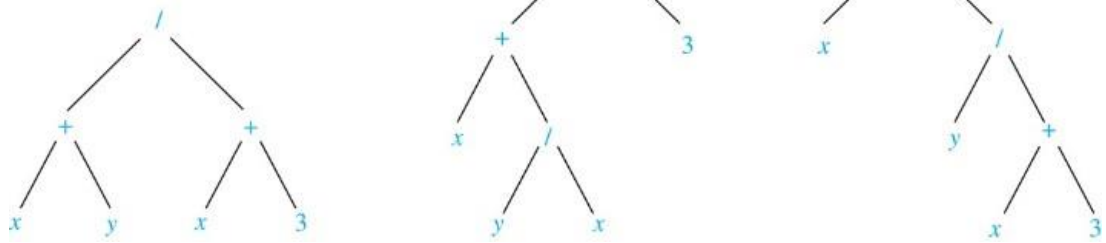


Jump to long description

Infix Notation

An inorder traversal of the tree representing an expression produces the original expression when parentheses are included except for unary operations, which now immediately follow their operands.

We illustrate why parentheses are needed with an example that displays three trees all yield the same infix representation.



Jump to long description

Prefix Notation

Jan Łukasiewicz (1878-1956)

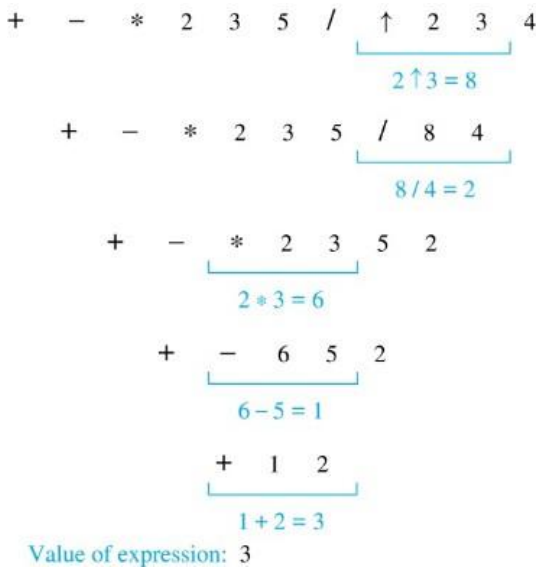
When we traverse the rooted tree representation of an expression in preorder, we obtain the *prefix* form of the expression. Expressions in prefix form are said to be in *Polish notation*, named after the Polish logician Jan Łukasiewicz.

Operators precede their operands in the prefix form of an expression. Parentheses are not needed as the representation is unambiguous.

The prefix form of $((x + y) \uparrow 2) + ((x - 4)/3)$ is $+ \uparrow + x\ y\ 2\ / - x\ 4\ 3$.

Prefix expressions are evaluated by working from right to left. When we encounter an operator, we perform the corresponding operation with the two operations to the right.

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



Jump to long description

Postfix Notation

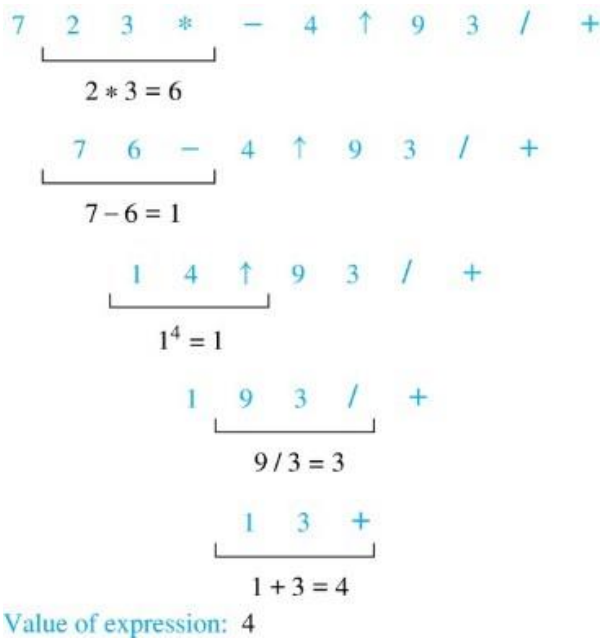
We obtain the *postfix form* of an expression by traversing its binary trees in postorder. Expressions written in postfix form are said to be in *reverse Polish notation.*

Parentheses are not needed as the postfix form is unambiguous.

$x\ y + 2 \uparrow x\ 4 - 3 / +$ is the postfix form of $((x + y) \uparrow 2) + ((x - 4)/3)$.

A binary operator follows its two operands. So, to evaluate an expression one works from left to right, carrying out an operation represented by an operator on its preceding operands.

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



Jump to long description

Spanning Trees

Section 11.4

Section Summary 3

Spanning Trees

Depth-First Search

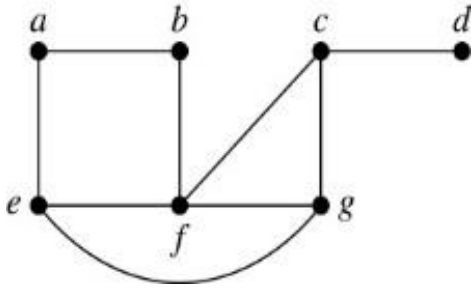
Breadth-First Search

Backtracking Applications (*not currently included in overheads*)

Depth-First Search in Directed Graphs

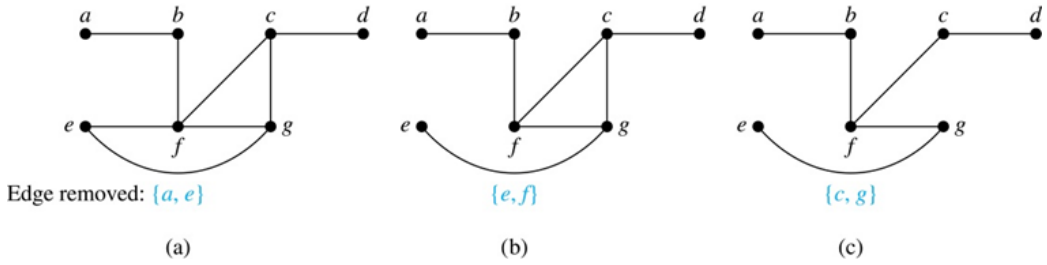
Spanning Trees 1

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.



Spanning Trees 2

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*.

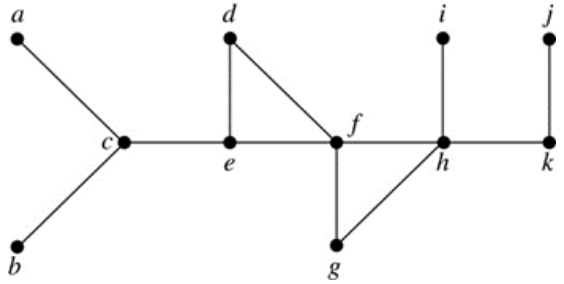
Depth-First Search 1

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.

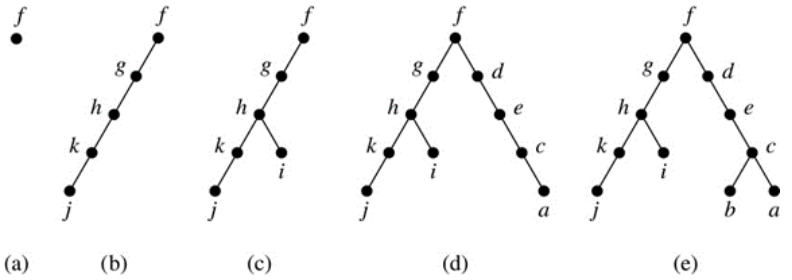
- Form a path starting at this vertex by successively adding vertices and edges, where each new edge is incident with the last vertex in the path and a vertex not already in the path. Continue adding vertices and edges to this path as long as possible.
- If the path goes through all vertices of the graph, 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 Search 2

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.

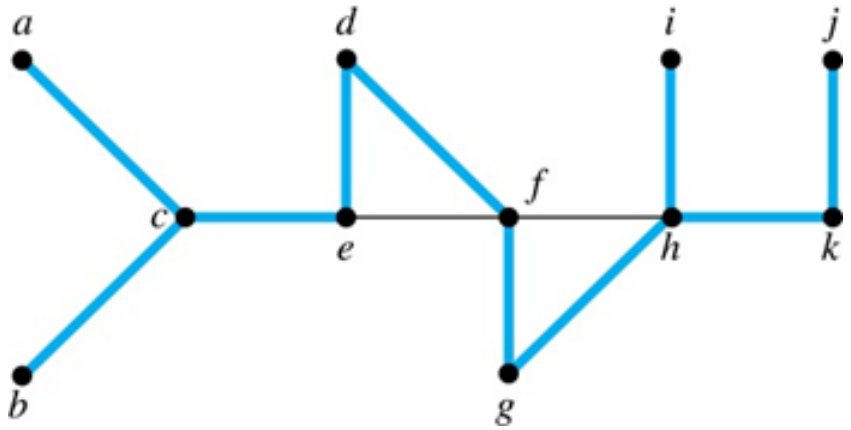


Jump to long description

Depth-First Search 3

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.

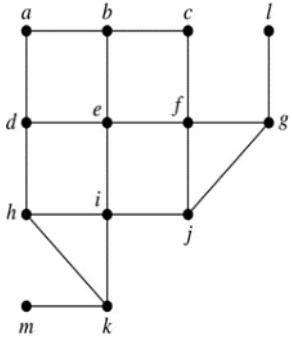
procedure *DFS*(G: connected graph with vertices v_1, v_2, …, v_n)
  T := tree consisting only of the vertex v_1
  visit(v_1)
procedure *visit*(v: vertex of G)
  for each vertex w adjacent to v and not yet in T
    add vertex w and edge $\{v,w\}$ to T
  visit(w)

Breadth-First Search 1

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

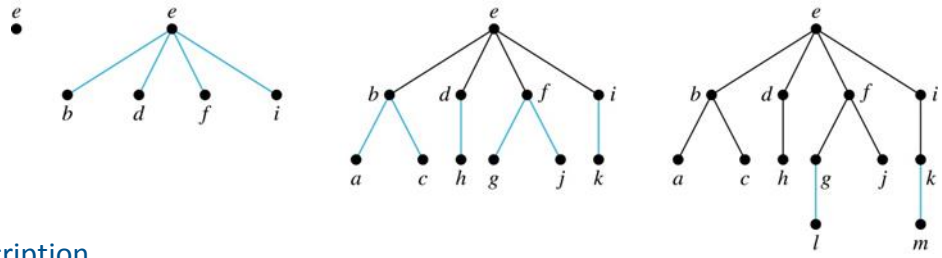
- Then we add all of the edges incident to this vertex and the other endpoint of each of these edges. We say that these are the vertices at level 1.
- For each vertex added at the previous level, we add each edge incident to this vertex, as long as it does not produce a simple circuit. The new vertices 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 Search 2



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

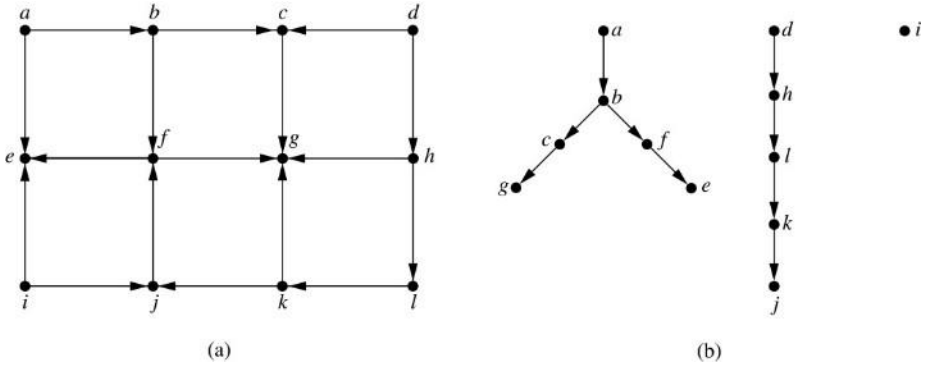
Breadth-First Search Algorithm

We now use pseudocode to describe breadth-first search.

procedure *BFS*(*G*: connected graph with vertices v_1, v_2, …, 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 breadth-first search or depth-first search to explore the Web graph.

Jump to long description

Appendix of Image Long Descriptions

Trees 1 – Appendix

All graphs have 6 vertices. A, B, C. D, E, and F. Graph G1 has 5 edges. A C, B C, D C. C F, and E F. Graph G2 has 5 edges. A C, B E, A F, D F, and E F. Graph G3 has 6 edges. A B, A C, A D. B E, D E, and C F. Graph G4 has 4 edges. A F, B D, B E, and C E.

Jump to the image

Trees as Models – Appendix

Middle image: The root is the root directory /. Internal vertices are directories. Leaves are files. The root has 3 edges that lead to USR, BIN, and TMP. Node USR has 3 edges that lead to BIN, RJE, and SPOOL. Node BIN has 3 edges that lead to ED, NROFF, and VI. Node RJE has 1 edge that leads to KHR. Node SPOOL has 2 edges that leads to OPR and UUCP. Node OPR has 1 edge that leads to Printer. Node UUCP has 1 edge that leads to File. Node BIN has 3 edges that lead to ls, Mail, and Who. Node TMP has 1 edge that leads to Junk.

Bottom image: The root is President and it has 4 edges that lead to VP R&D, VP Marketing, VP Services, and VP Finance. Node VP R&D has 3 edges that lead to Director Research, Director Software Development, and Director Hardware Development. Node VP Marketing has 2 edges that lead to AVP Sales and AVP Marketing. Node VP Services has 2 edges that lead to Chief Field Operations and Director Material Management. Node VP Finance has 2 edges that lead to Director Accounting and Director MIS.

Jump to the image

Rooted Trees – Appendix

All graphs have 7 vertices. A, B, C. D, E, F, and G. The first graph, labeled T, has 6 edges. C E, C A, A D. A B, B F, and B G. Root C is at the bottom. The second graph has 6 edges. A B, B F, B G. A C, C E, and A D. Root A is at the top. The edges are directed from the top to the bottom. The third graph has 6 edges. C E, C A, A D. A B, B G, and B F. Root C is at the top. The edges are directed from the top to the bottom.

Jump to the image

Rooted Tree Terminology – Appendix

The tree has 10 vertices and its depth is 3. The root node is Nikolaus 1623-1708. It has 3 edges that lead to Jacob I 1654-1705, Nikolaus 1662-1716, and Johann I 1667-1748. Nikolaus has 1 edge that leads to Nikolaus I 1687-1759. Johann I has 3 edges that lead to Nikolaus II 1695-1726, Daniel 1700-1782, and Johann II 1710-1790. Johann II has 2 edges that lead to Johann III 1746-1807 and Jacob II 1759-1789.

Jump to the image

Terminology for Rooted Trees – Appendix

The graph has 13 vertices. A, B, C. D, E, F. G, H, I. J, K, L, and M. The graph has 12 edges. A B, A F, A G. B C, C D, C E. G H, G I, G J. H K, J L, and J M.

Jump to the image

m-ary Rooted Trees – Appendix

The first graph has 7 vertices and 6 edges. Each internal node has 2 children. The second graph has 13 vertices and 12 edges. Each internal node has 3 children. The third graph has 16 vertices and 15 edges. Each internal node has 5 children. The fourth graph has 11 vertices and 10 edges. Internal nodes have 2-3 children.

Jump to the image

Ordered Rooted Trees – Appendix

Tree T has 13 vertices. A, B, C. D, E, F. G, H, I. J, K, L, and M. The graph has 12 edges. A B, B D, B E. D F, D G, C H, H J. C I, I K. I L, and L M. Graph B has 2 vertices, H and J, connected by an edge. Graph C has 4 vertices. I, K, L, and M. The graph has 3 edges. I K, I L, and L M.

Jump to the image

Level of vertices and height of trees – Appendix

The graph has 13 edges. A B, A J, A K. B C, B E, B F. C D, F G, F I. G H, K L, L M, and L N.

Jump to the image

Balanced *m*-Ary Trees – Appendix

T1 has 20 vertices and 19 edges. Its height is 4, and its leaves are at levels 3 and 4. T2 has 13 vertices and 12 edges. Its height is 4, and its leaves are at levels 2, 3, and 4. The third tree has 20 vertices and 19 edges. Its height is 3, and its leaves are at level 3.

Jump to the image

Preorder Traversal$_1$ – Appendix

The root of the tree is labeled R. It has N children labeled T1 to T N. The preorder traversal begins by visiting R. It continues by traversing T1 in preorder, then T2 in preorder, and so on, until T N is traversed in preorder.

Jump to the image

Preorder Traversal 2 – Appendix

By the preorder traversal algorithm, first visit the root and then visit the subtrees from left to right. The first step shows node A, subtree B, node C, and subtree D. The second step shows nodes A and B, subtree E, nodes F, C, and D. Subtree G, and nodes H and I. The third step shows nodes A, B, E, J, subtree K, and nodes from F to I. The fourth step is the preorder traversal of T. It shows nodes from A to I from left to right.

Jump to the image

Inorder Traversal$_1$ – Appendix

The root of the tree is labeled R. It has N children labeled T1 to T N. The preorder traversal begins by traversing T1 in inorder, then visiting R. It continues by traversing T2 in inorder, and so on, until T N is traversed in inorder.

Jump to the image

Inorder Traversal 2 – Appendix

The tree has 16 vertices labeled from A to P. A is the root. The tree has 15 edges. A B, A C, A D. B E, B F, E J, E K. K N, K O, K P. D G, D H, D I. G L, and G M. By the inorder traversal algorithm, first visit leftmost subtree, then visit root, and then visit other subtrees left to right. The first step shows subtree B, nodes A and C, and subtree D. The second step shows subtree E, nodes B, F, A, C. Subtree G, nodes G, H, and I. The third step shows nodes J and E, subtree K. Nodes B, F, A, C, L. G, M, D, H, and I. The fourth step is the inorder traversal of T. It shows vertices J, E, N. K, O, P. B, F, A. C, L, G. M, D, H, and I.

Jump to the image

Postorder Traversal$_1$ – Appendix

The root of the tree is labeled R. It has N children labeled T1 to T N. The postorder traversal begins by traversing from T1 in postorder to T N in postorder, and ends by visiting R.

Jump to the image

Postorder Traversal$_2$ – Appendix

The tree has 16 vertices labeled from A to P. A is the root. The tree has 15 edges. A B, A C, A D. B E, B F, E J, E K. K N, K O, K P. D G, D H, D I. G L, and G M. By the postorder traversal algorithm, first visit subtrees left to right and then visit root. The first step shows subtree B, node C, subtree D, and node A. The second step shows subtree E, nodes F, B, C. Subtree G, nodes H, I, D, and A. The third step shows node J, subtree K, nodes E, F, B. C, L, M, G. H, I, D, and A. The fourth step is the postorder traversal of T. It shows nodes J, N, O. P, K, E. F, B, C. L, M, G. H, I, D, and A.

Jump to the image

Expression Trees – Appendix

The first subtree represents the summation of X and Y. There is root + with two children, X and Y. The second subtree represents the subtraction of X and 4. There is root - with two children, X and 4. The third subtree represents the exponentiation of X + Y to 2. There is root an upward arrow with two children, + and 2. Node + has two children, X and Y. The fourth subtree represents the division of X - 4 by 3. There is root / with two children, -, and 3. Node - has two children, X and 4. The binary tree represents the whole expression: X + Y, squared plus, X - 4, divided by 3.

Jump to the image

Infix Notation – Appendix

The root of the first tree is / and it has 2 children, + and +. The first + has 2 children, X and Y. The second + has 2 children, X and 3. The root of the second tree is + and it has 2 children, + and 3. Node + has 2 children, X and /. Node / has 2 children, Y and X. The root of the third tree is + and it has 2 children. X and /. Node / has 2 children, Y and +. Node + has 2 children, X and 3.

Jump to the image

Prefix Notation – Appendix

The first step is evaluating exponentiation 2 3. The value is 8. The second step is evaluating divide 8 4. The value is 2. The third step is evaluating multiply 2 3. The value is 6. The fourth step is evaluating minus 6 5. The value is 1. The fifth step is evaluating plus 1 2. The value is 3. The value of prefix expression is 3.

Jump to the image

Postfix Notation – Appendix

The first step is evaluating 2 3 multiply. The value is 6. The second step is evaluating 1 6 minus. The value is 1. The third step is evaluating 1 4 exponentiation. The value is 1. The fourth step is evaluating 9 3 divide. The value is 3. The fifth step is evaluating 1 3 plus. The value is 4. The value of postfix expression is 4.

Jump to the image

Depth-First Search 2 – Appendix

Step 1 shows vertex F. Step 2 shows a graph with vertices F, G, H. K, and J. And 4 edges. F G, G H, H K, and K J. Step 3 shows the same graph as in step 2 with an additional vertex I and edge H I. Step 4 shows the same graph as in step 3 with an additional vertices D, E, C, and A. And 4 edges. F D, D E, E C, and C A. Step 5 shows the same graph as in step 4 with an additional vertex B and edge C B.

Jump to the image

Breadth-First Search 2 – Appendix

Top image: The graph has 18 edges. A B, B C, A D. B E, C F, L G. D E, E F, F G. D H, E I, F J. G J, H I, I J. H K, I K, and K M.

Bottom image: The first step is vertex E. The second step is adding vertices B, D, F, I. And 4 edges. E B, E D, E F, and E I. The third step is adding vertices A, C, H. G, J, and K. And 6 edges B A, B C, D H, F G, F J, and I K. The fourth step is adding vertices L and M. And 2 edges, G L and K M.

Jump to the image

Depth-First Search in Directed Graphs – Appendix

The graphs have 12 vertices labeled from A to L. The arrows of graph A point from A to B, from B to C, from D to C. From A to E, from B to F, from C to G. From D to H, from H to G, from F to G. From F to E, from I to E, from J to F. From K to G, from H to L, from L to K. From K to J, and from I to J. Graph B is disconnected and its arrows point from A to B, from B to C, from C to G. From B to F, from F to E. From D to H, from H to L, from L to K, and from K to J.

Jump to the image