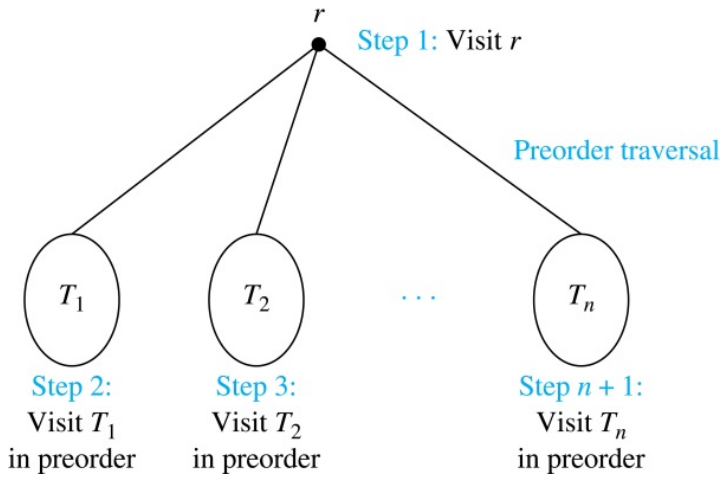
11.3 Tree Traversal

Tree Traversal

- A traversal algorithm is a procedure for systematically visiting every vertex of an ordered rooted tree.
- Tree traversals are defined recursively.
- Three traversals are named:
  - ✓ preorder,
  - ✓ inorder,
  - ✓ postorder.

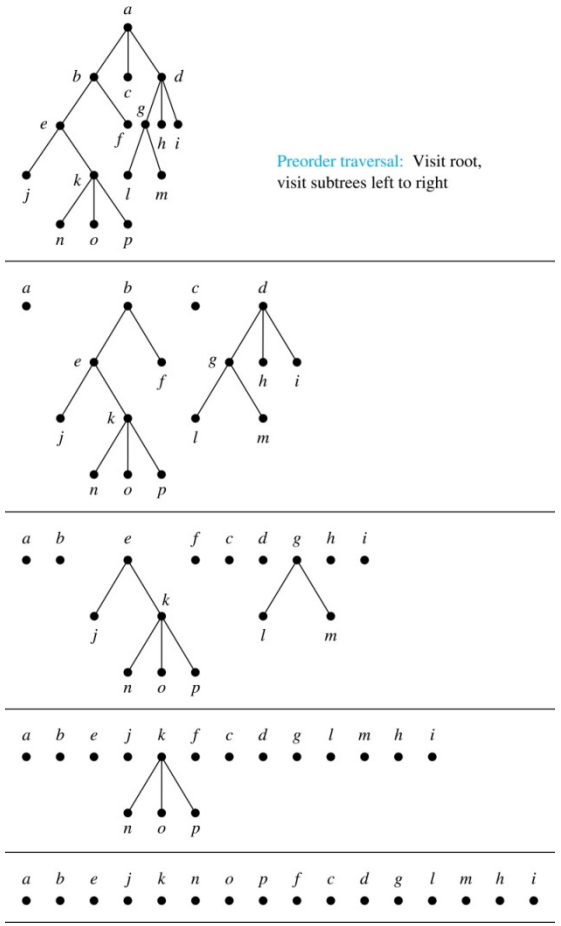
Preorder Traversal

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



Preorder Traversal (continued)

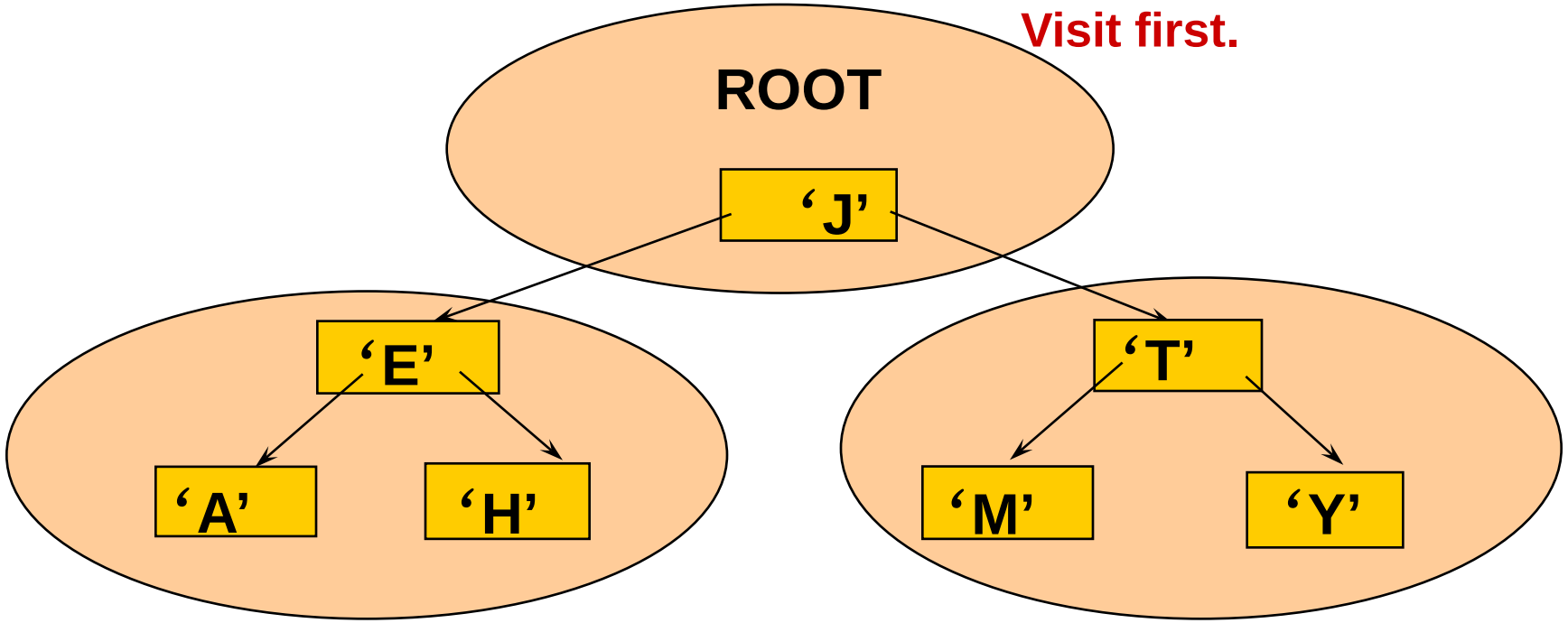
procedure *preorder* (*T*: ordered rooted tree)
r* := root of *T
list *r*
for each child *c* of *r* from left to right
    ***T*(*c*) := subtree with *c* as root**
    ***preorder*(*T*(*c*))**



Preorder traversal of an binary ordered tree

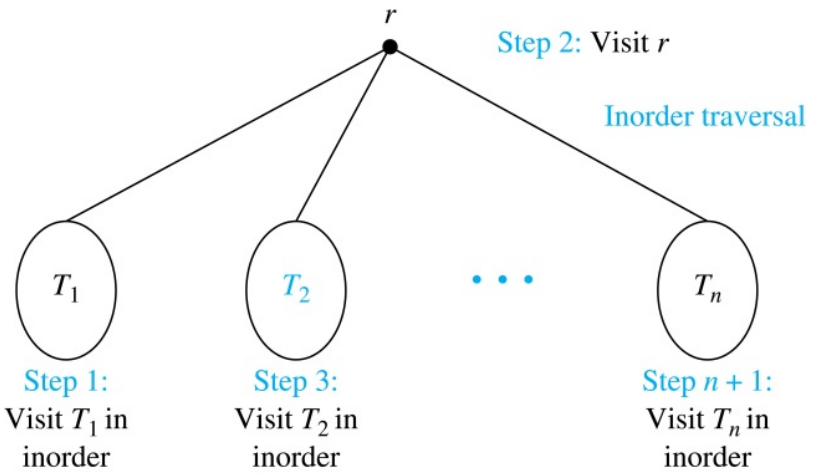
- Visit the root.
- Visit the left subtree, using preorder.
- Visit the right subtree, using preorder.

Preorder Traversal: J E A H T M Y



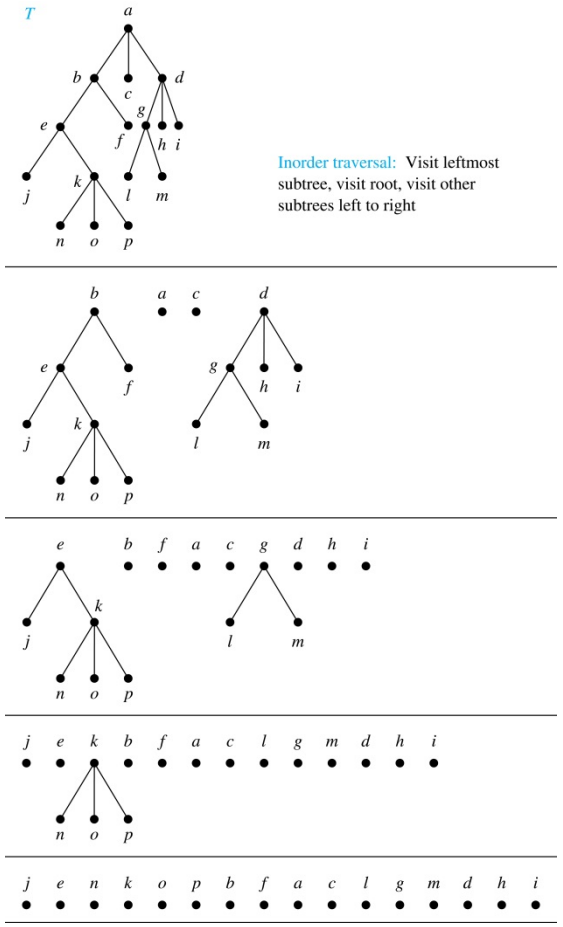
Inorder Traversal

Definition: Let T be an ordered tree with root r. If T has only r, then r is the inorder traversal of T. Otherwise, suppose T_1, T_2,…, T_n are the left to right subtrees at r. The inorder traversal begins by traversing T_1 in inorder. Then visits r, then traverses T_2 in inorder, and so on, until T_n is traversed in inorder.



Inorder Traversal (continued)

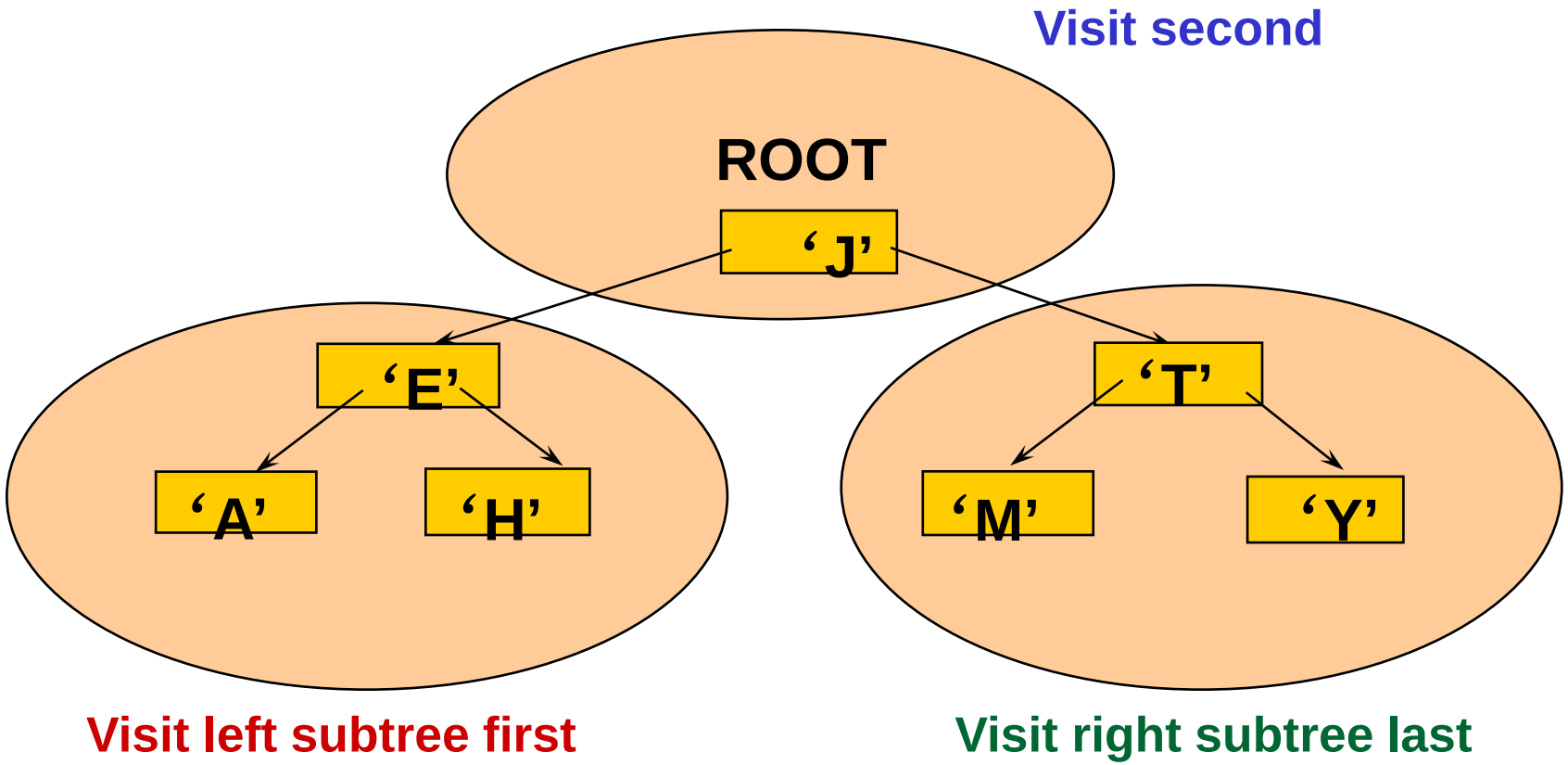
procedure ***inorder*** **(*T*: ordered rooted tree)**
r **:= root of *T***
if *r* is a leaf then list *r*
else
    l **:= first child of *r* from left to right**
    T**(*l*) := subtree with *l* as its root**
    inorder**(*T*(*l*))**
    list(*r*)
    for each child *c* of *r* from left to right
 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



Inorder traversal of an binary ordered tree

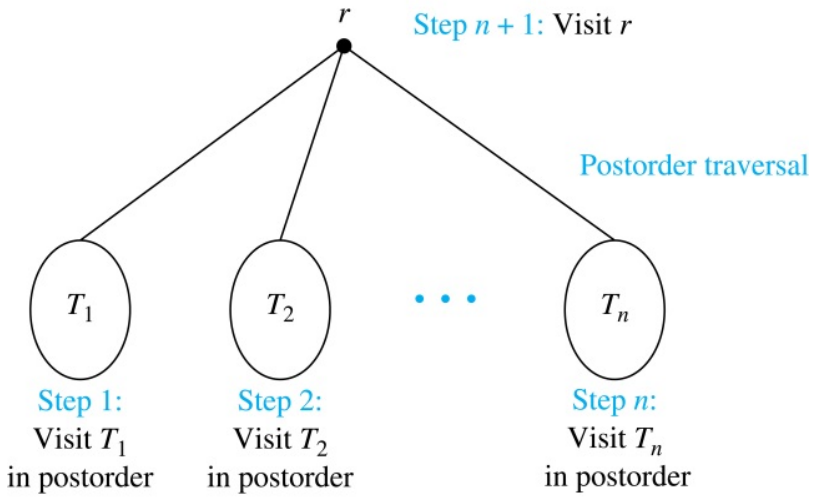
- **Visit the left subtree, using inorder.**
- **Visit the root.**
- **Visit the right subtree, using inorder.**

Inorder Traversal: A E H J M T Y



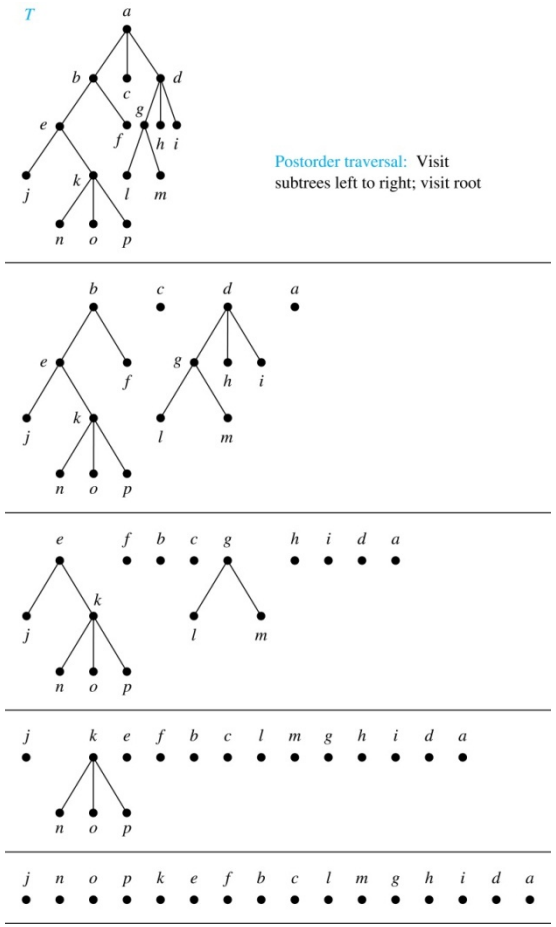
Postorder Traversal

Definition: Let T be an ordered tree with root r.If T has only r, then r is the postorder traversal of T. Otherwise, suppose T_1, T_2,…, T_n are the left to right subtrees at r. The postorder traversal begins by traversing T_1 in postorder. Then traverses T_2 in postorder, until T_n is traversed in postorder, finally ends by visiting r.



Postorder Traversal (continued)

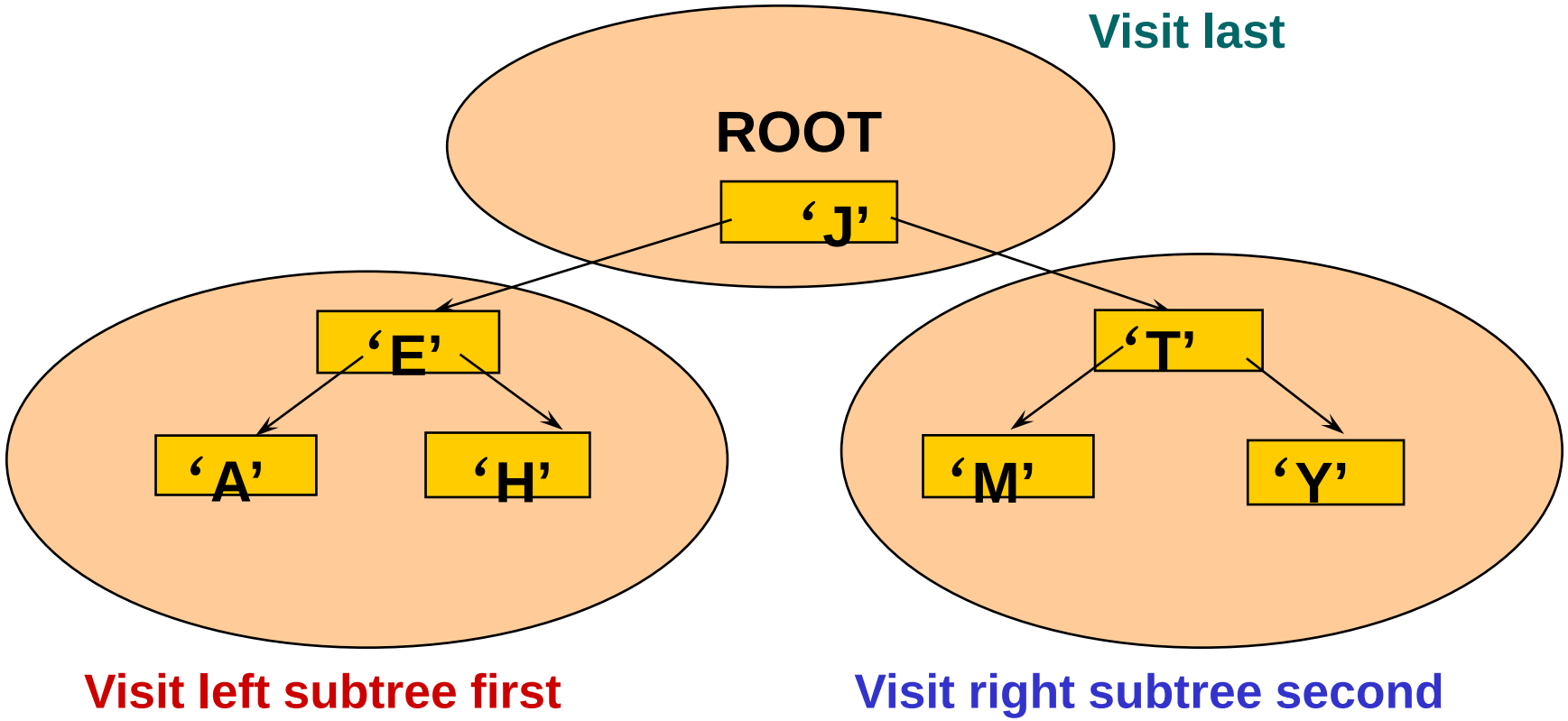
```
procedure  postordered (T: ordered rooted tree)
r := root of T
for each child c of r from left to right
    T(c) := subtree with c as root
    postorder(T(c))
list r
```



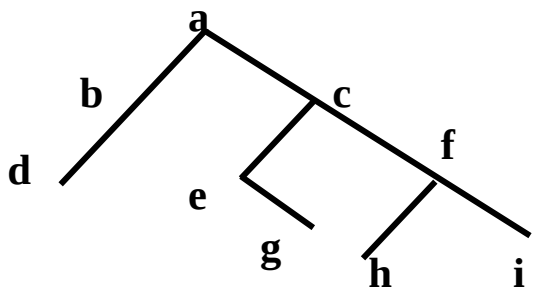
Postorder traversal of an binary ordered tree

- Visit the left subtree, using postorder.
- Visit the right subtree, using postorder.
- Visit the root.

Postorder Traversal: A H E M Y T J



〖Example 1〗 In which order does a preorder, inorder or postorder traversal vist the vertices in the ordered rooted tree shown in the following figure?



- Preorder traversal : a, b, d, c, e, g, f, h, i
- Inorder traversal : d, b, a, e, g, c, h, f, i
- Postorder traversal : d, b, g, e, h, i, f, c, a

Expression Trees

Complicated expressions can be represented using ordered rooted trees, such as

- ✓ **Compound propositions**
- ✓ **Combinations of sets**
- ✓ **Arithmetic expressions**

A Binary Expression Tree is a special kind of binary tree in which:

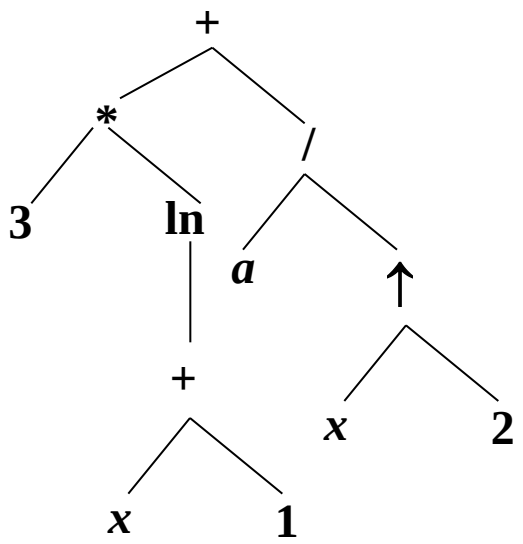
- Each leaf node contains a single operand,
- Each nonleaf node contains a single operator, and
- The left and right subtrees of an operator node represent subexpressions that must be evaluated before applying the operator at the root of the subtree.

How to build an expression tree

〖Example 2〗 What is the ordered tree that represents the expression $3*\ln(x+1)+a/x^2$?

Solution:

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



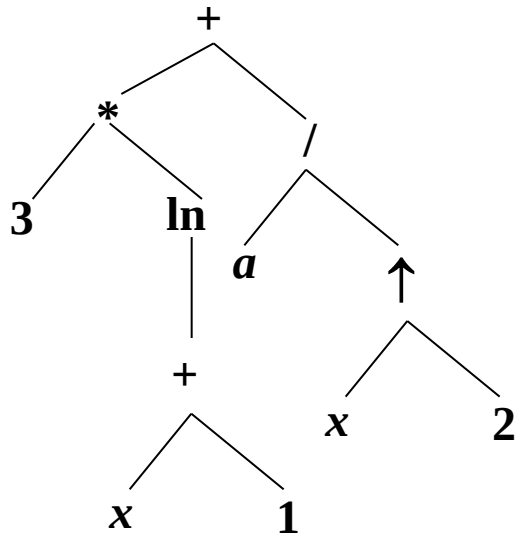
How to evaluate ?

〖Example 2〗 What is the ordered tree that represents the expression $3*\ln(x+1)+a/x^2$?

Infix Form

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.

For example,

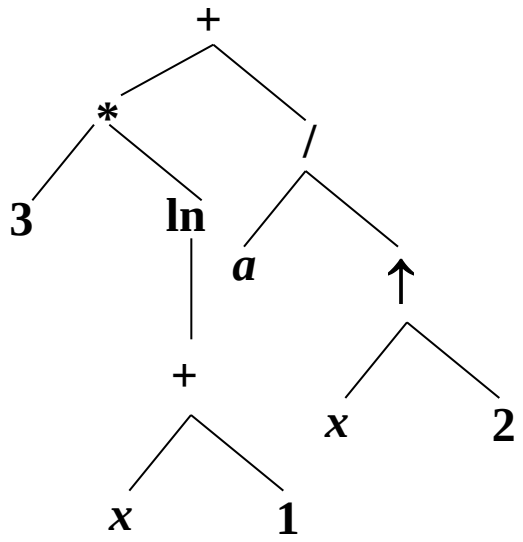


Infix form: $3 * \ln(x+1) + a / x \uparrow 2$

Prefix Form

The expression obtained by an preorder traversal of the binary tree is said to be in prefix form (Polish notation).

For example,



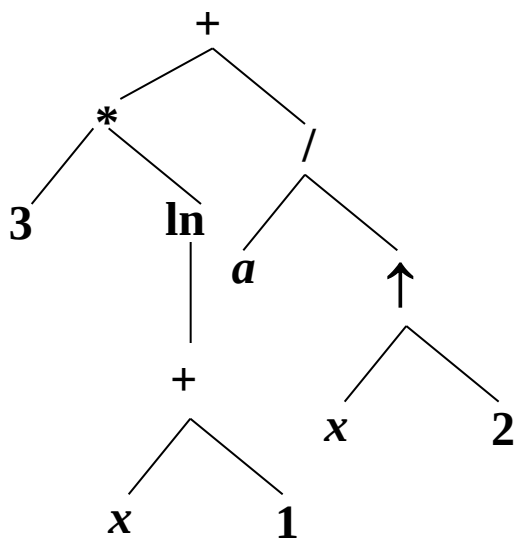
Prefix form: $+*3\ln+x1/a\uparrow x2$

- **Operators precede their operands in the prefix form of an expression.**
- **Parentheses are not needed as the representation is unambiguous.**
- **Prefix expressions are evaluated by working from right to left.**

Postfix Form

The expression obtained by an postorder traversal of the binary tree is said to be in postfix form (reverse Polish notation).

For example,



- **Parentheses are not needed as the postfix form is unambiguous.**
- **to evaluate an expression one works from left to right.**

Postfix form: $3x1+\ln{*}ax2\uparrow/+$

Homework:

SE：P. 783　8, 16

EE：P. 819　8, 16

11.4-11.5 Spanning Trees & Minimum Spanning Trees

Spanning Tree

- The definition of spanning tree
- How to find a spanning tree of the simple graph
- The application of spanning tree
- Algorithms for constructing spanning trees
- Backtracking

The definition of spanning tree

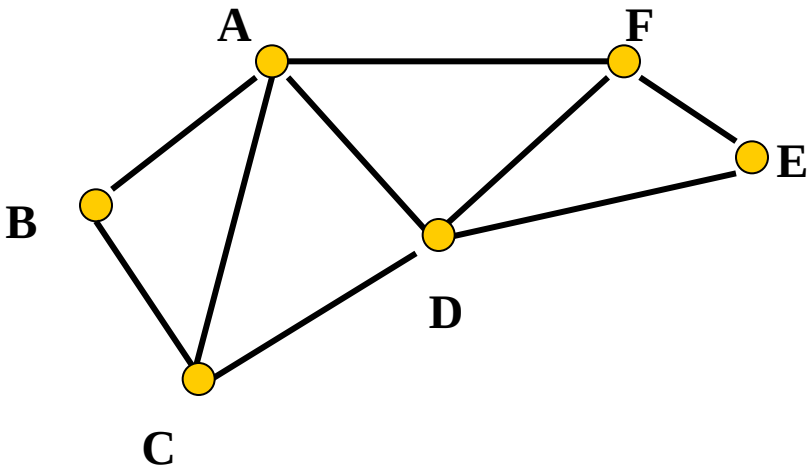
【Definition 1】 Let G be a simple graph. A spanning tree of G is a subgraph of G that is a tree containing every vertex of G.

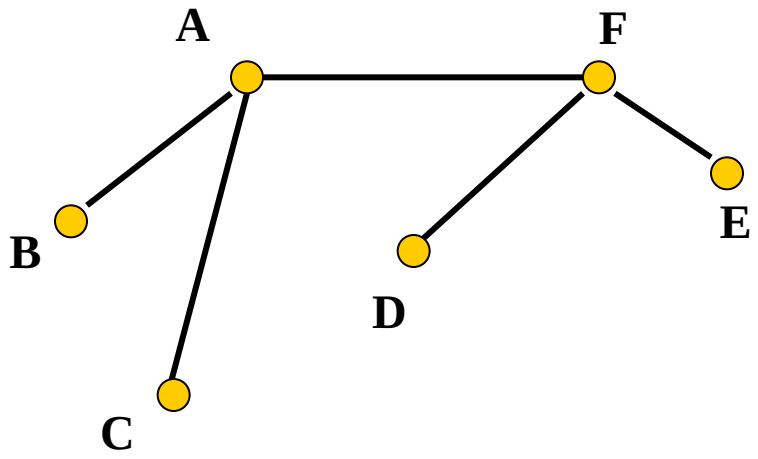
For example,

Problem:

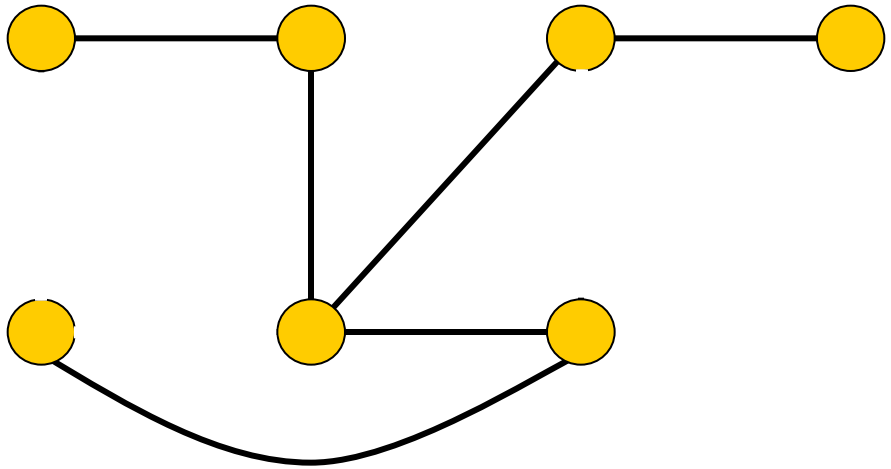
Why should we study the problem of spanning tree?

-- Consider the system of roads





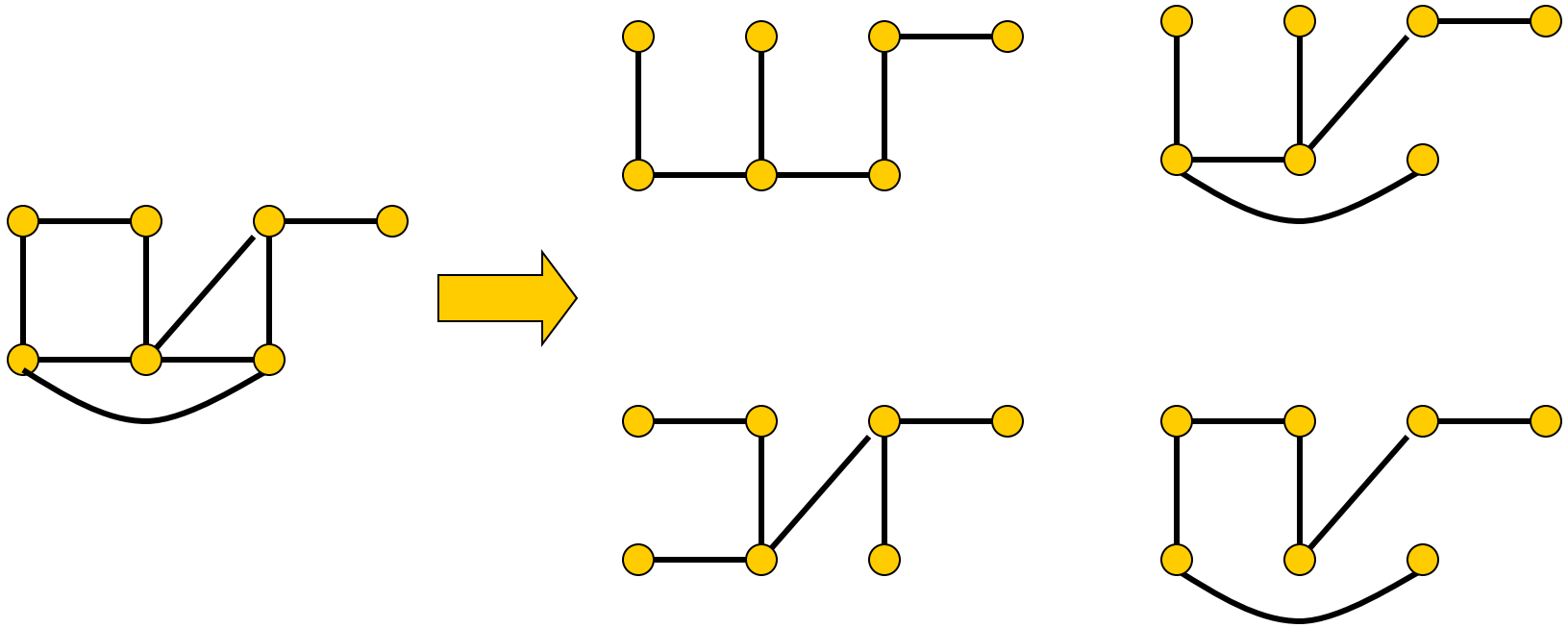
Find A Spanning Tree of The Simple Graph



Method:

Find spanning trees by removing edges from simple circuits.

More than one spanning tree for a simple graph



【 Theorem 1 】 A simple graph is connected if and only if it has a spanning tree.

Proof:

First, suppose that a simple graph *G* has a spanning tree tree *T*.

***T* contains every vertex of *G*.**

There is a path in *T* between any two of its vertices.

Since T is a subgraph of *G*, there is a path in *G* between any two of its vertices.

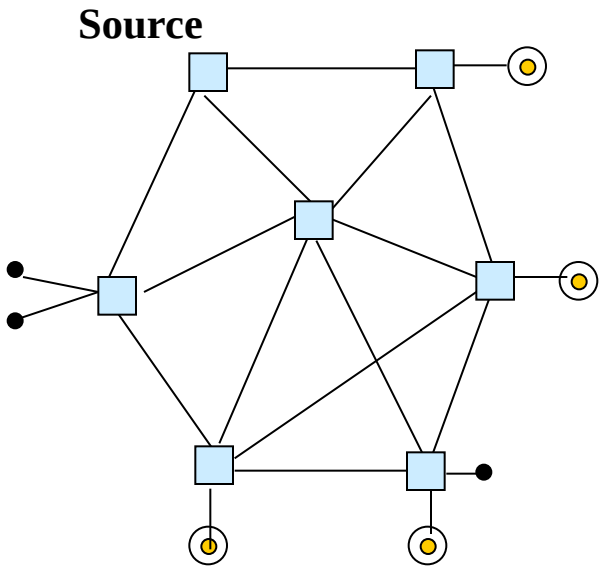
Hence *G* is connected.

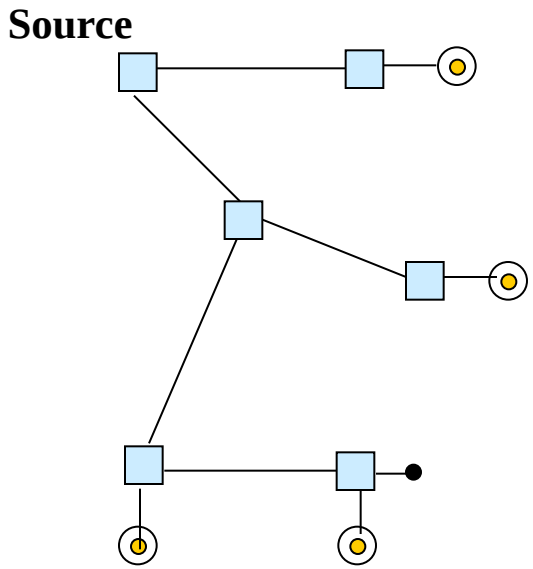
Second, suppose that *G* is connected.

We can find a spanning trees by removing edges from simple circuits of G.

The Application of Spanning Tree

〖Example 1〗 IP Multicasting.





Algorithms for constructing spanning trees

- Theorem 1 gives an algorithm for finding spanning trees by removing edges from simple circuits.
- Instead of constructing spanning trees by removing edges, spanning trees can be built up by successively adding edges.
- Two algorithm:
  - ✓ Depth-first search
  - ✓ Breadth-first search

Depth-first search

Depth-first search (also called **backtracking**) -- this procedure forms a rooted tree, and the underlying undirected graph is a spanning tree.

1. Arbitrarily choose a vertex of the graph as root.
2. From a path starting at this vertex by successively adding edges, where each new edge is incident with the last vertex in the path and a vertex not already in the path.
3. Continue adding edges to this path as long as possible.
4. If the path goes through all vertices of the graph, the tree consisting of this path is a spanning tree.
5. If the path does not go through all vertices, more edges must be added. Move back to the next to last vertex in the path, if possible, form a new path starting at this vertex passing through vertices that were not already visited. If this cannot be done, move back another vertex in the path.
6. Repeat this process.

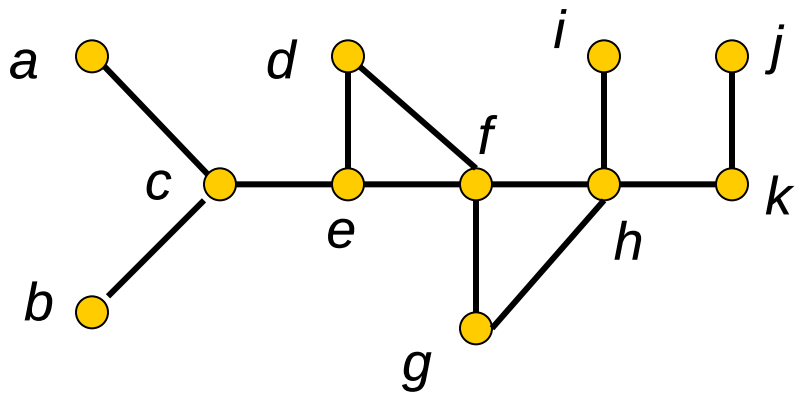
Depth-First Search Algorithm

- 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.

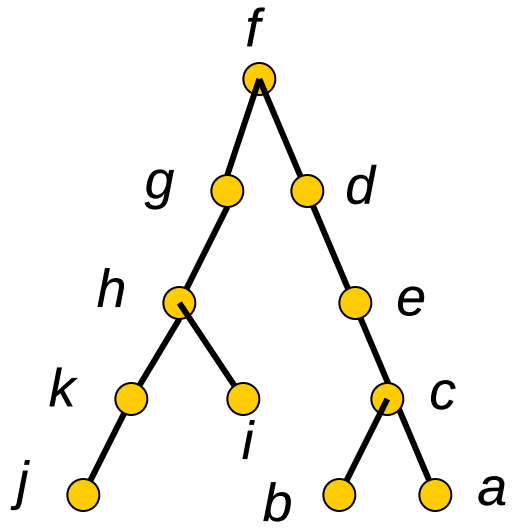
procedure *DFS*(*G*: connected graph with vertices $v_1, v_2, \ldots, v_n$)
***T* := tree consisting only of the vertex v_1**
***visit*(v_1)**

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*)**

〖Example 2〗 Use a depth-first search to find a spanning tree for the following graph.

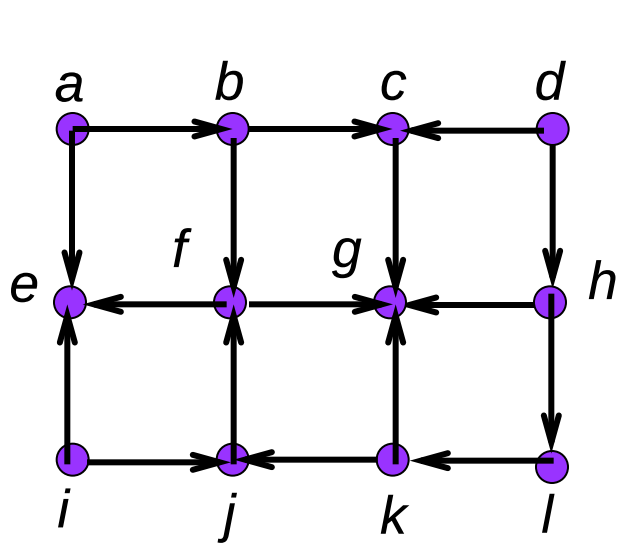


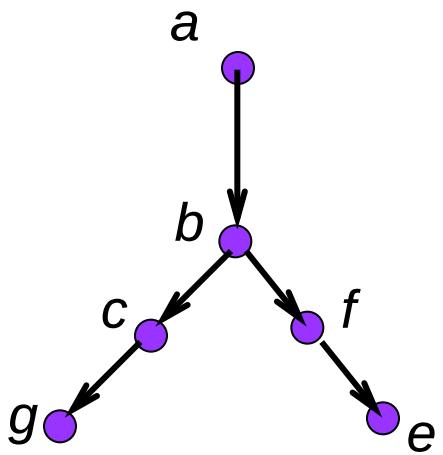
Solution:

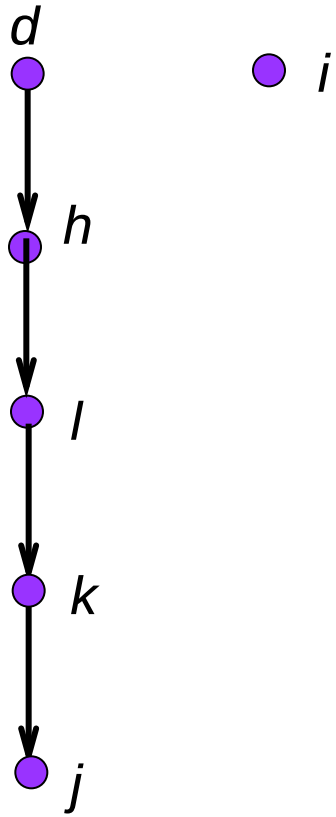


Depth-first Search in Directed Graphs

〖Example 6〗 What is the output of depth-first search given the following graph as input?







Breadth-first search

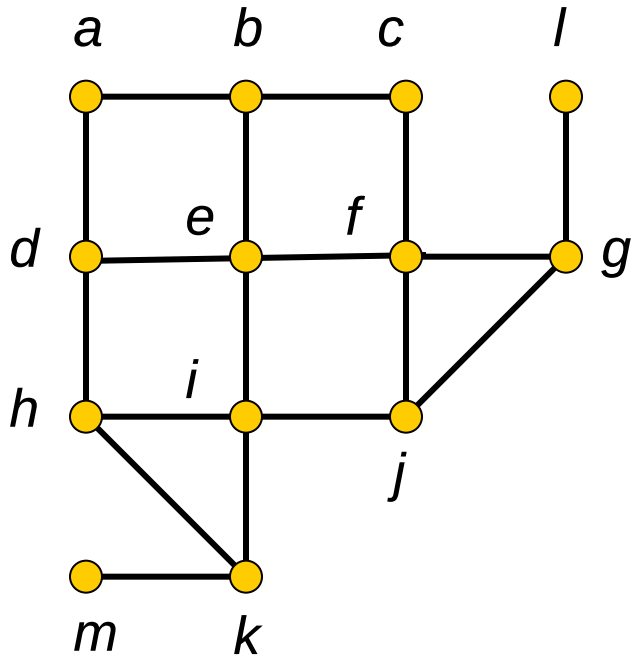
1. Arbitrarily choose a vertex of the graph as a root, and add all edges incident to this vertex.
2. The new vertices added at this stage become the vertices at level 1 in the spanning tree. Arbitrarily order them.
3. For each vertex at level 1, visited in order, add each edge incident to this vertex to the tree as long as it does not produce a simple circuit. Arbitrarily order the children of each vertex at level 1. This produces the vertices at level 2 in the tree.
4. Follow the same procedure until all the vertices in the tree have been added.

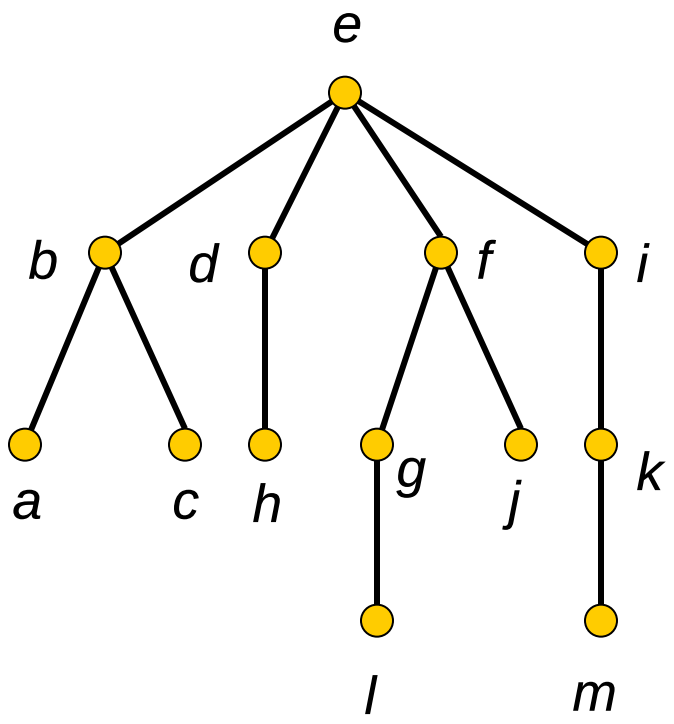
Breadth-First Search Algorithm

- We now use pseudocode to describe breadth-first search.

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

〖Example 3〗 Use a breadth-first search to find a spanning tree for the following graph.





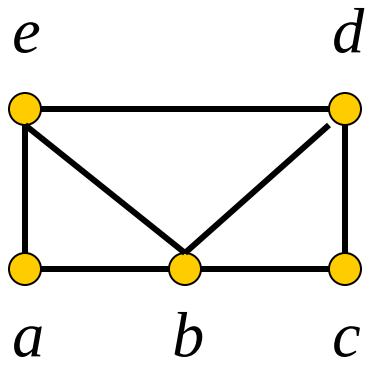
Backtracking scheme

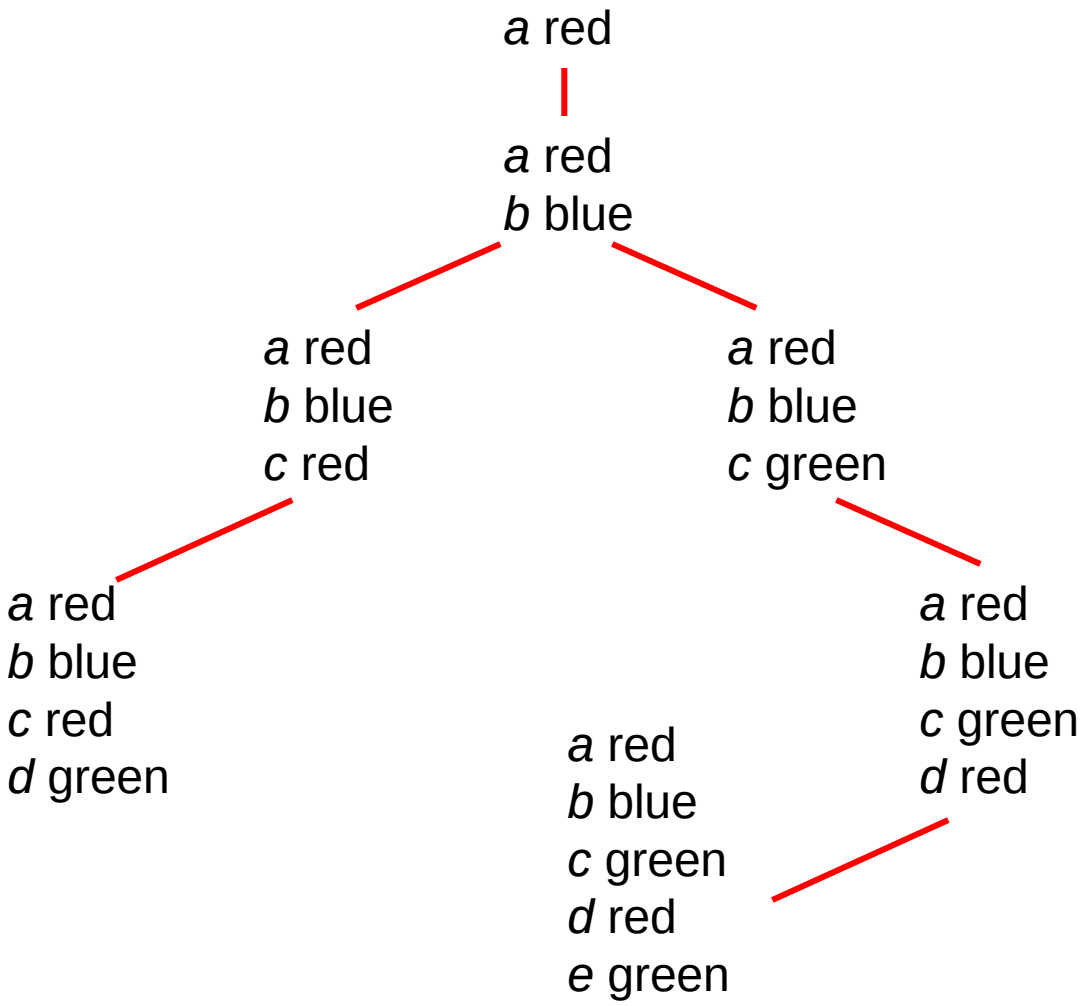
- What is backtracking ?
- The application of backtracking scheme
  - ✓ Graph Coloring
  - ✓ The n-Queens Problem
  - ✓ Sums of Subsets

Backtracking scheme

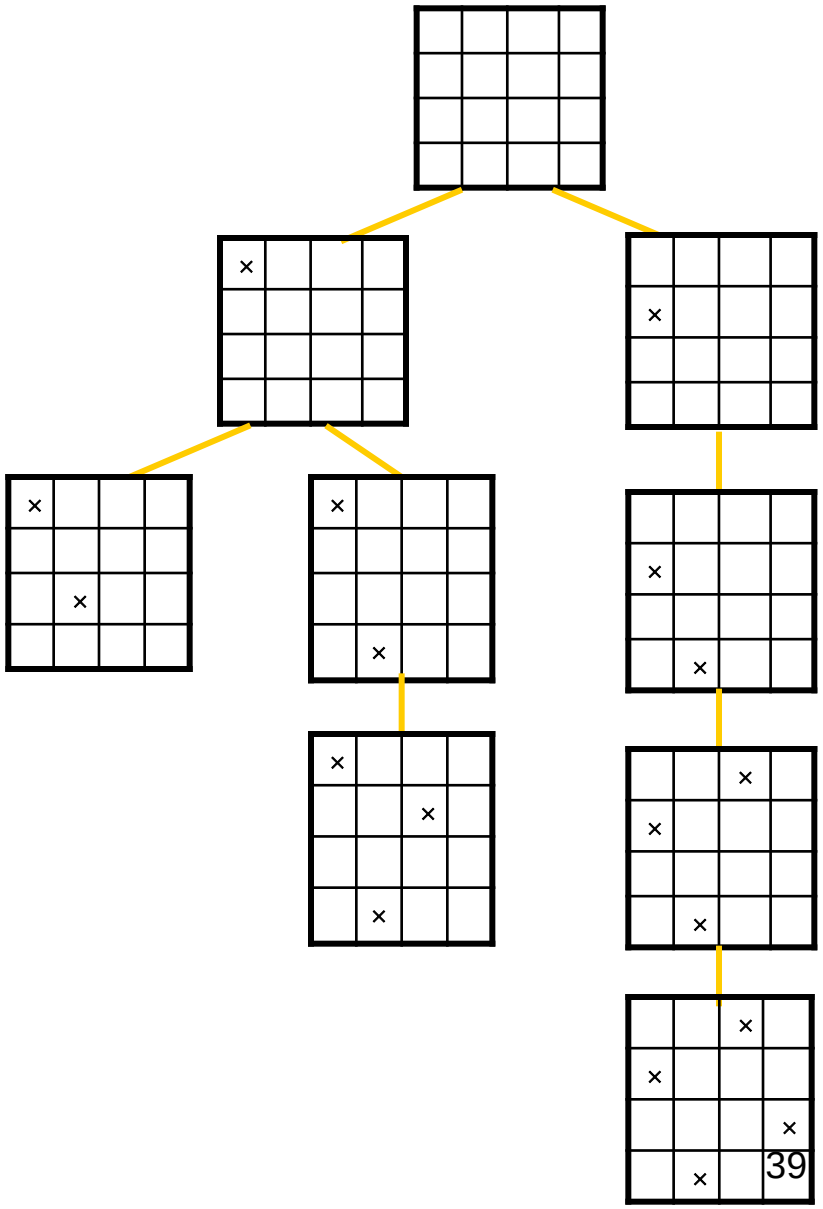
- There are problems that can be solved only by performing an exhaustive search of all possible solutions.
- One way to search systematically for a solution is to use a decision tree, where each internal vertex represents a decision and each leaf a possible solution.
- The method to find a solution via backtracking
- The application of backtracking scheme
  - ✓ Graph Coloring
  - ✓ The n-Queens Problem
  - ✓ Sums of Subsets

〖Example 4〗 How can backtracking be used to decide whether the following graph can be colored using 3 colors?

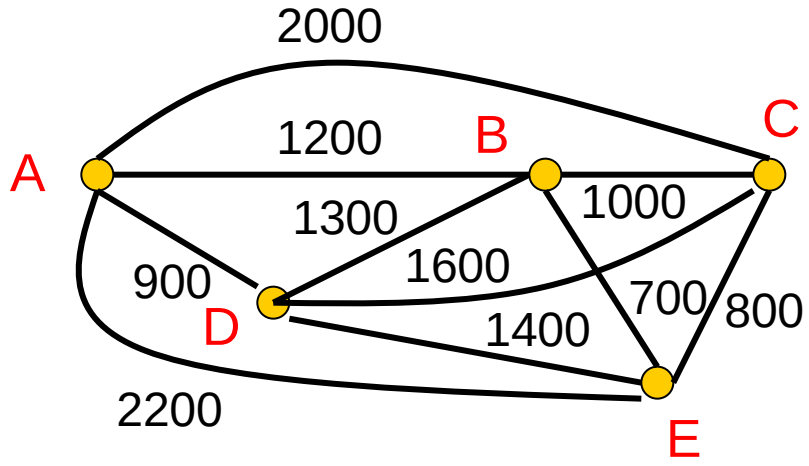




〖Example 5〗 How can backtracking be used to solve the *n*-queens problem?



the Concept of Minimum Spanning Trees



A weighted graph showing monthly lease costs for lines in a computer network.

Problem:

Which links should be made to ensure that there is a path between any two computer centers so that the total cost of the network is minimized?

We can solve this problem by finding a spanning tree so that the sum of the weights of the edges of the tree is minimized. Such a spanning tree is called a minimum spanning tree.

the Concept of Minimum Spanning Trees

【Definition 1】 A minimum spanning tree in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

Algorithms for minimum spanning trees

Two algorithms for constructing minimum spanning trees:

- ✓ **Prim's algorithm**
- ✓ **Kruskal's algorithm**

Both proceed by successively adding edges of smallest weight from those edges with a specified property that have not already been used.

These two algorithms are examples of greedy algorithms.

Prim's algorithm

Procedure ***Prim*** **(*G*: weighted connected undirected graph with *n* vertices)**

***T*:= a minimum-weight edge**

for *i*:= 1 to *n*-2

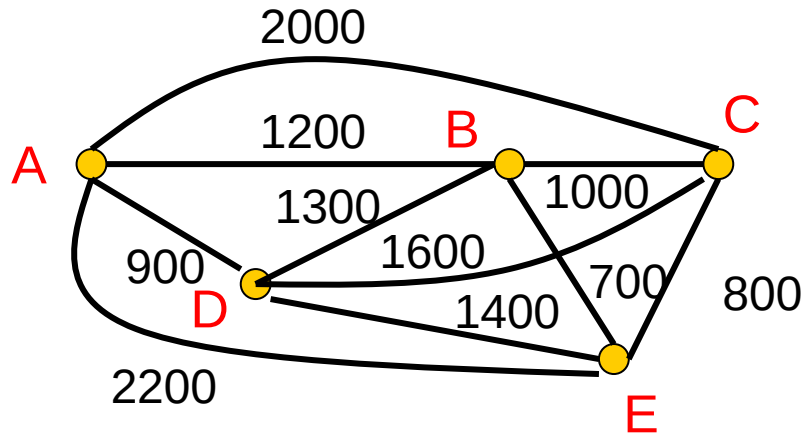
begin

***e*:= an edge of minimum weight incident to a vertex in *T* and not forming a simple circuit in *T* if added to *T*.**

***T*:= *T* with *e* added**

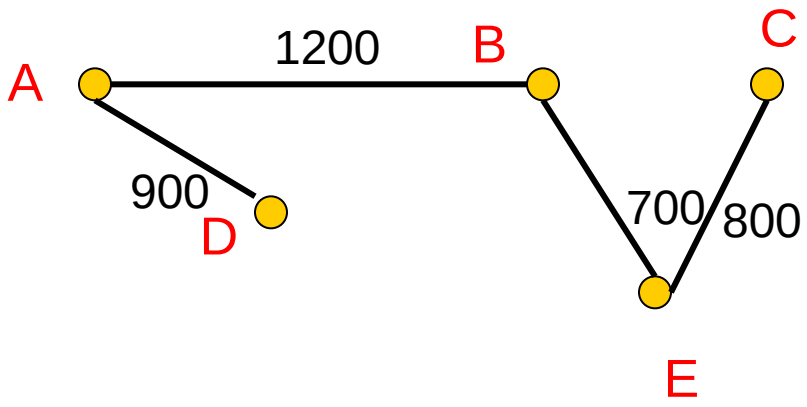
end {*T* is a minimum spanning tree of *G*}

〖Example 1〗 Find a minimum spanning tree in the weighted graph.



Solution:

A minimum spanning tree and the choices of edges at each stage of Prim's algorithm are shown in the following graph.



Choice	Edge	Cost
1	BE	700
2	EC	800
3	BA	1200
4	AD	900
	Total:	3600

Kruskal's algorithm

procedure Kruskal (*G*: weighted connected undirected graph with n vertices)

T:= empty graph

for i:= 1 to *n*-1

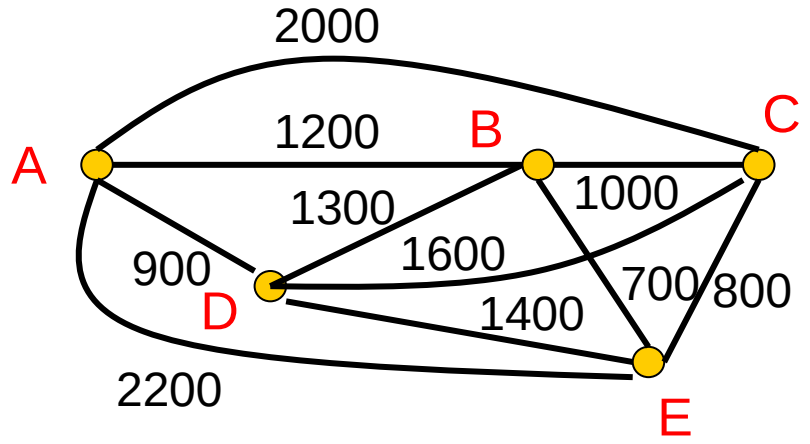
begin

e:= any edge in *G* with smallest weight that does not form a simple circuit when added to *T*

T:= *T* with e added

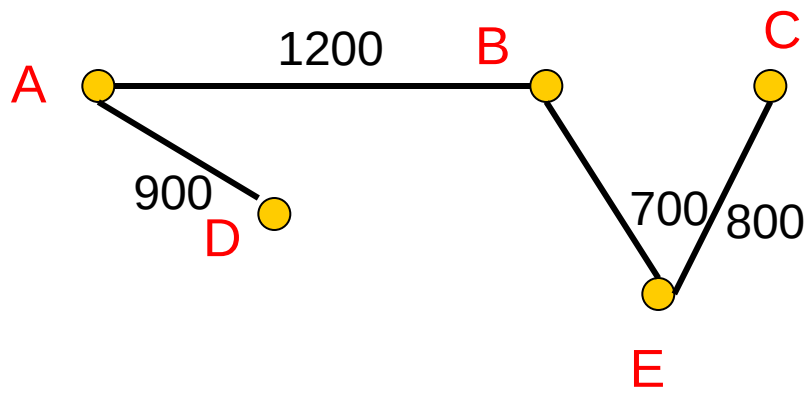
end {*T* is a minimum spanning tree of *G*}

〖Example 2〗 Find a minimum spanning tree in the following weighted graph.



Solution:

A minimum spanning tree and the choices of edges at each stage of Kruskal's algorithm are shown in the following graph.



Choice	Edge	Cost
1	BE	700
2	EC	800
3	AD	900
4	AB	1200
	Total:	3600

Homework:

SE：P. 795 4, 14, 16(14), 29

P. 802 3,7,12

EE：P. 832 4, 14, 16(14), 29

P. 839 3,7,12