VCU, Department of Computer Science

CMSC 302

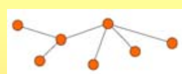
Trees

Vojislav Kecman

Topics

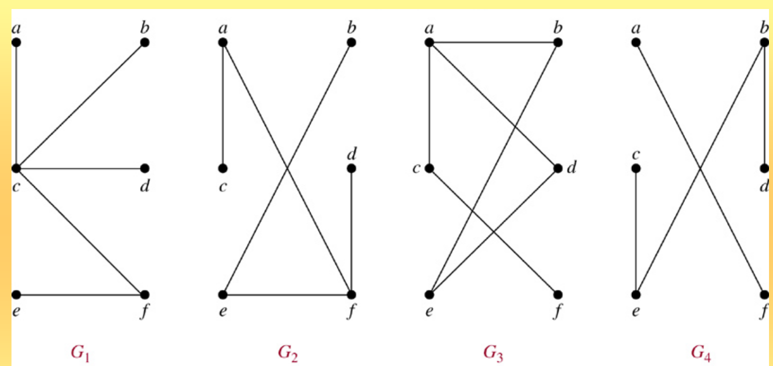
- Terminology
- Trees as Models
- Some Tree Theorems
- Applications of Trees
  - Binary Search Tree
  - Decision Tree
- Tree Traversal
- Spanning Trees

Terminology



- **Tree**
  - A *tree* is a **connected undirected graph** that contains **no circuits**.
  - **Theorem: There is a unique simple path between any two of its nodes.**
- A (not-necessarily-connected) undirected graph without simple circuits is called a ***forest*.**
  - You can think of it as a set of trees having disjoint sets of nodes
- Subtree of node (i.e., vertex) *n*
  - A tree that consists of a child (if any) of **node *n*** and the child's descendants
- Parent of node *n*
  - The node directly above **node *n*** in the tree
- Child of **node n**
  - A node directly below **node *n*** in the tree

Which graphs are trees?



- G1 and G2 are. G3 has circuits. G4 is not connected

Terminology

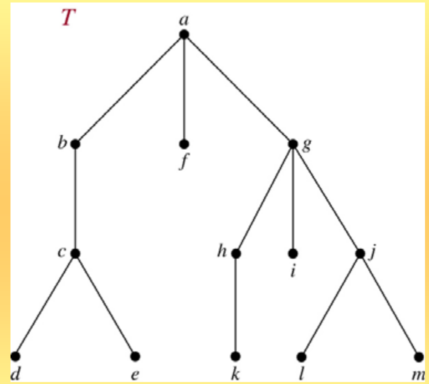
- **Root**
  - The **only node** in the tree **with no parent**
- **Leaf**
  - A node with no children
- **Siblings**
  - Nodes with a common parent
- **Ancestor** of node *n*
  - A node on the path from the root to *n*
- **Descendant** of node *n*
  - A node on a path from *n* to a leaf

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

We'll come back to this math a little latter

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

What are the relations/terms/connections for some vertices?

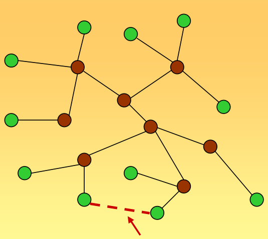


1. b internal?
2. k internal?
3. g subtree root?
4. k descendant of g?
5. d sibling of e?

Tree and Forest Examples

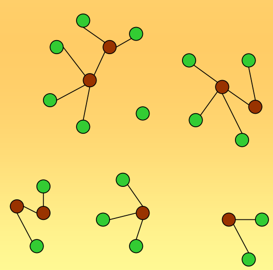
- A Tree:

Leaves in **green**,
internal nodes in **brown**.



Note: by adding this link, trees becomes graph. It's no longer tree!

- A Forest:

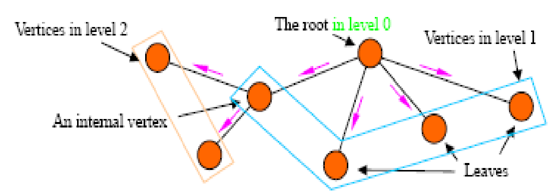


Rooted Trees

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. Vertices of degree one are called *leaves*; other vertices (including the root) are called *internal* vertices. The *level* of a vertex 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 levels of vertices.

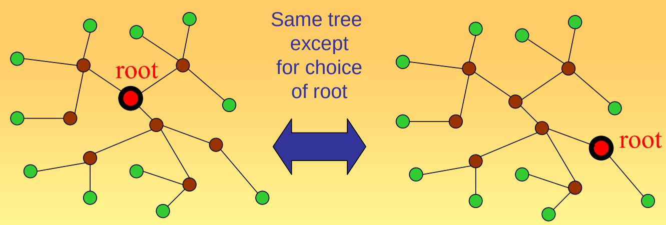
Example

A rooted tree of height 2

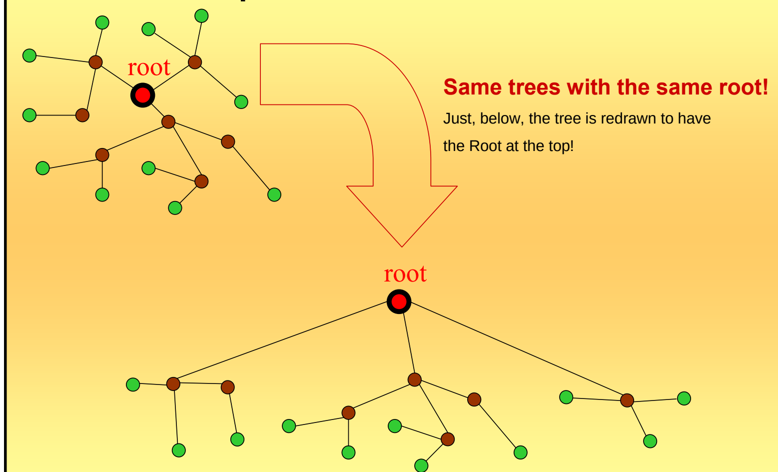


Examples of Rooted Trees

- Note that a given unrooted tree with *n* nodes yields *n* different rooted trees.



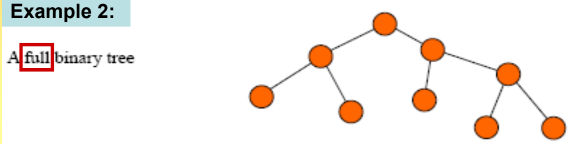
Examples of Rooted Trees



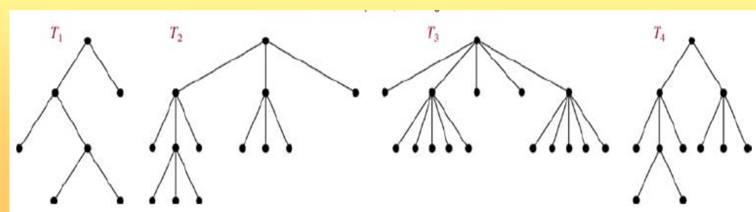
n-ary trees

- A tree is called ***n*-ary** if every vertex has **no more than *n* children.**
  - It is called ***full*** if every **internal** (non-leaf) vertex has *exactly n* children.
- A **2-ary** tree is called a ***binary tree***.
  - These are handy **for describing sequences of yes-no decisions.**
    - **Example 1:** Comparisons in binary search algorithm.

Example 2:



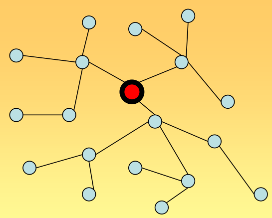
What trees are shown below?



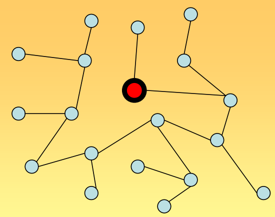
T_1 full binary T_2 full 3-ary T_3 full 5-ary T_4 3-ary

Which Tree is Binary?

- **Theorem:** A given rooted tree is a binary tree iff every node other than the root has **degree ≤ 3**, and the **root has degree ≤ 2.**



NO (the root has degree 3)



YES

Some algebraic properties of trees

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

Proof Since each internal vertex of a full m-ary tree has m children, there should be mi vertices of the tree except the root. The total number of a full m-ary tree is $mi + 1$.

Corollary There are $(m-1)i + 1$ leaves in a full m-ary tree.

Proof The number of leaves in a full m-ary tree is equal to the total number of vertices minus the number of internal vertices; hence there are $mi + 1 - i = (m-1)i + 1$ leaves.

On the next slide, there will be some repetition and a little more of the trees' theory !

Some Tree Theorems

- Any tree with n nodes has $e = n-1$ edges.
- A full m-ary tree with i internal nodes has $n=mi+1$ nodes, and $\ell=(m-1)i+1$ leaves.
  - **Proof:** There are mi children of internal nodes, plus the root. And, $\ell = n-i = (m-1)i+1$. □
- Thus, when m is known and the tree is full, we can compute all four of the values e, i, n, and ℓ, given any one of them.

More algebras about FULL *m*-ary trees

Full m-ary tree with:

(i) n vertices has

$i=(n-1)/m$ internal vertices

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

(ii) i internal vertices has

$n=mi+1$ vertices

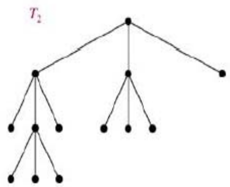
$l=(m-1)i+1$ leafs.

(iii) l leafs has

$n=(ml-1)/(m-1)$ vertices

$i=(l-1)/(m-1)$ internal vertices.

Example:



Full 3-ary tree:

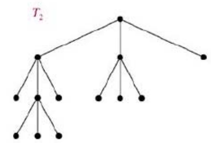
l = 9 leafs

n = (3*9-1)/(3-1) = 13 total vertices *(iii) l leafs has n=(ml-1)/(m-1)* vertices

i = (9-1)/(3-1) = 4 internal vertices *(iii) i=(l-1)/(m-1)* internal vertices

l = (3-1)*4 + 1 = 9 leafs *(ii) l=(m-1)i+1* leafs.

One more example



Chain letter sent to 3 others represented by 3-ary tree with leaf vertices those not sending letter.

1. How many people have *seen* letter, including first person, if no duplicates and letter ends after 100 people received but did not send? Total vertices those who have seen letter.
2. How many people *sent* the letter? Internal vertices those who sent letter.

l=100 Leaf vertices, people receiving but not sending the letter.

(iii) l leafs = 100 *m-ary = 3*

n=(ml-1)/(m-1) vertices

i=(l-1)/(m-1) internal vertices.

By (iii) n=(3*100-1)/(3-1) = 299/2 = 149 vertices, the number who have *seen* the letter.

By (iii) i=(100-1)/(3-1) = 99/2 = 49 internal vertices, the number who *sent* out the letter.

Some More Tree Theorems

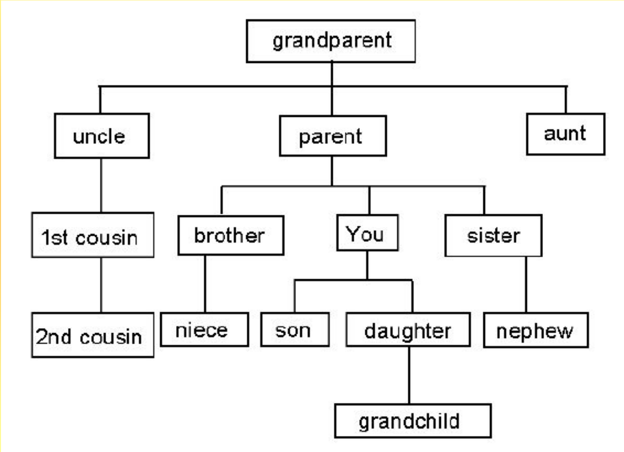
- **Definition:** The *level* of a node is the length of the simple path from the root to the node.
  - The *height* of a tree is maximum node level.
  - A rooted *m*-ary tree with height *h* is called *balanced* if all leaves are at levels h or $h-1$.
- **Theorem:** There are at most m^h leaves in an *m*-ary tree of height *h*.
  - **Corollary:** An *m*-ary tree with ℓ leaves has height $h \geq \lceil \log_m \ell \rceil$. If *m* is full and balanced then $h = \lceil \log_m \ell \rceil$.

Trees as Models

- Can use trees to model the following:
  - Saturated hydrocarbons
  - Organizational structures
  - Computer file systems
    - See about these in your textbook
    - Perform the self learning here – it's simple

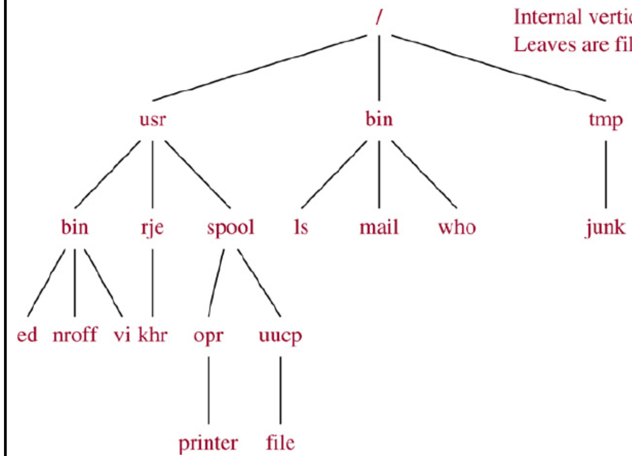
Two examples follow!

Tree as the model



Another one!

The root is the root directory /
Internal vertices are directories
Leaves are files

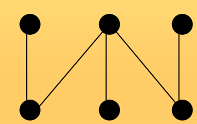


10.2 Applications of Trees

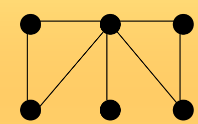
– Search trees
– Decision trees
– Prefix codes
– Expression trees

Before proceeding any further a tiny quiz for you? Which graphs are trees?

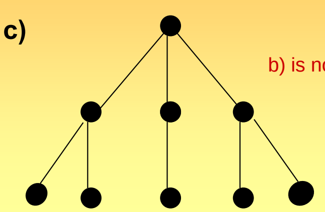
a)



b)



c)



b) is not a tree, for there are circuits

Searching always takes time; for huge relational data, and/or for huge trees, and/or for huge data sets, it takes huge time

So the goal in computer programs is often to find any stored item efficiently when all stored items are ordered.

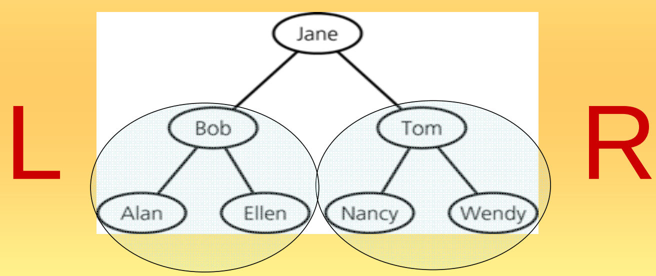
A Binary Search Tree can be used to store items in its vertices. It enables efficient searches.

A Binary Search Tree (BST) is . . .

A special kind of binary tree in which:

1. **Each vertex contains a distinct key value,**
2. **The key values in the tree can be compared using "greater than" and "less than", and**
3. **The key value of each vertex in the tree is less than every key value in its right subtree, and greater than every key value in its left subtree.**

Binary Search Trees



Shape of a binary search tree . . .

Depends on its key values and their order of insertion.

Insert the elements 'J' 'E' 'F' 'T' 'A' in that order.

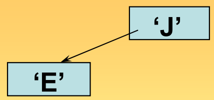
The first value to be inserted is put into the root.

'J'

Inserting ‘E’ into the BST

‘J’ ‘E’ ‘F’ ‘T’ ‘A’

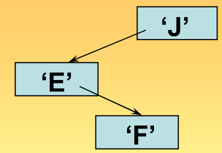
Thereafter, each value to be inserted begins by comparing itself to the value in the root, moving left if it is less, or moving right if it is greater. This continues at each level until it can be inserted as a new leaf.



Inserting ‘F’ into the BST

‘J’ ‘E’ ‘F’ ‘T’ ‘A’

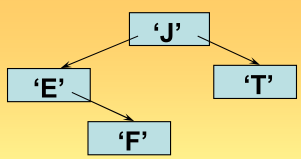
Begin by comparing ‘F’ to the value in the root, moving left if it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.



Inserting ‘T’ into the BST

‘J’ ‘E’ ‘F’ ‘T’ ‘A’

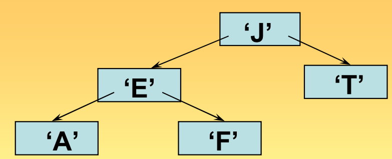
Begin by comparing ‘T’ to the value in the root, moving left if it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.



Inserting ‘A’ into the BST

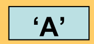
‘J’ ‘E’ ‘F’ ‘T’ ‘A’

Begin by comparing ‘A’ to the value in the root, moving left if it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.



What binary search tree . . .

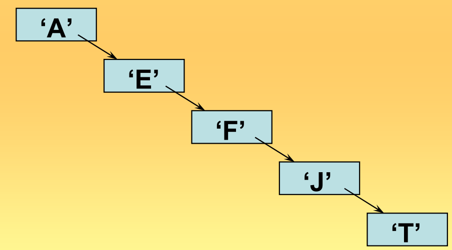
is obtained by inserting
the elements ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ in that order?



We are inserting ‘A’ ‘E’ ‘F’ ‘J’ ‘T’

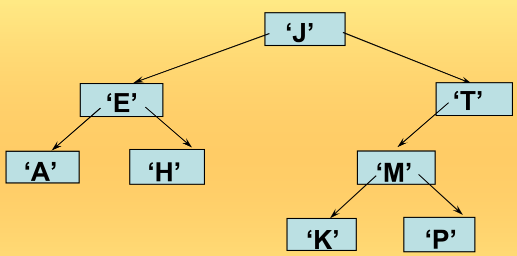
Binary search tree . . .

obtained by inserting
the elements ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ in that order.



We are inserting ‘A’ ‘E’ ‘F’ ‘J’ ‘T’

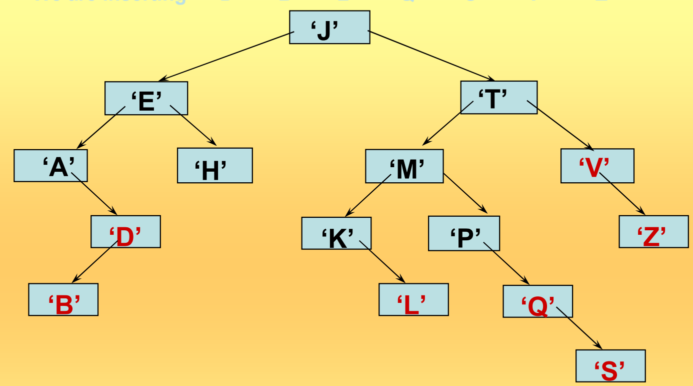
Another binary search tree



Add nodes containing these values in this order:

‘D’ ‘B’ ‘L’ ‘Q’ ‘S’ ‘V’ ‘Z’

We are inserting ‘D’ ‘B’ ‘L’ ‘Q’ ‘S’ ‘V’ ‘Z’



How would you go about searching whether ‘F’ is in the binary search tree?

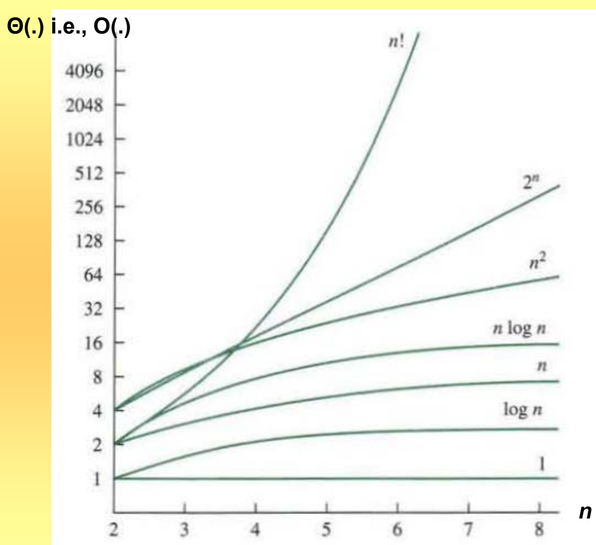
Binary Search Trees (BST) properties

- BST supports the following operations in Θ(log n) i.e., O(log n) average-case time:
  - Searching for an existing item.
  - Inserting a new item, if not already present.
- BST supports printing out all items in Θ(n) time.
- Note that inserting into a plain sequence a_i would instead take Θ(n) worst-case time.

What is the meaning of Θ(.) i.e., O(.)?
(those are the famous Theta or Big O notations).

There is a traditional hierarchy of algorithms:

- **O(1)** is constant-time; such an algorithm does not depend on the size of its inputs.
- **O(n)** is linear-time; such an algorithm looks at each input element once and is generally pretty good.
- **O(n log n)** is also pretty decent (that is n times the logarithm base 2 of n).
- **O(n^2), O(n^3), etc.** These are polynomial-time, and generally starting to look pretty slow, although they are still useful.
- **O(2^n)** is exponential-time, which is common for machine learning, i.e., data mining tasks and is really quite bad. Exponential-time algorithms begin to run the risk of having a decent-sized input not finish before the person wanting the result retires.
- There are worse; like O(2^2^...(n times)...^2).



Recursive Binary Tree Insert

- **procedure** insert(T: binary tree, x: item)
  v := root[T]
  if v = null **then begin**
  root[T] := x; **return** “Done” **end**
  else if v = x **return** “Already present”
  else if x < v **then**
  return insert(leftSubtree[T], x)
  else {must be x > v}
  return insert(rightSubtree[T], x)

Decision Trees

- A decision tree represents a *decision-making process*.
  - Each possible “decision point” or situation is represented by a node.
  - Each possible choice that could be made at that decision point is represented by an edge to a child node.
- In the extended decision trees used in *decision analysis*, we also include nodes that represent random events and their outcomes.

Coin-Weighing Problem

- Imagine you have 8 coins, one of which is a lighter counterfeit, and a free-beam balance.
  - No scale of weight markings is required for this problem!
- How many weighings are needed to guarantee that the counterfeit coin will be found?

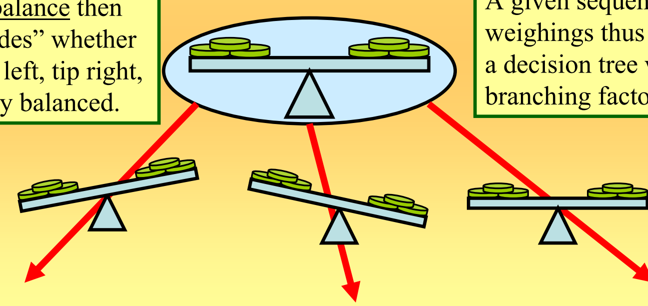
What is the trivial solutions in terms of the number of weighings?

- Obviously, in the worst case scenario, we can always find the counterfeit coin by making 4 pairwise weighing.
- In the best case scenario this approach may result in finding the counterfeit coin in the first measurement.

As a Decision-Tree Problem

- In each situation, we pick two disjoint and equal-size subsets of coins to put on the scale.

The balance then “decides” whether to tip left, tip right, or stay balanced.



A given sequence of weighings thus yields a decision tree with branching factor 3.

General Solution Strategy

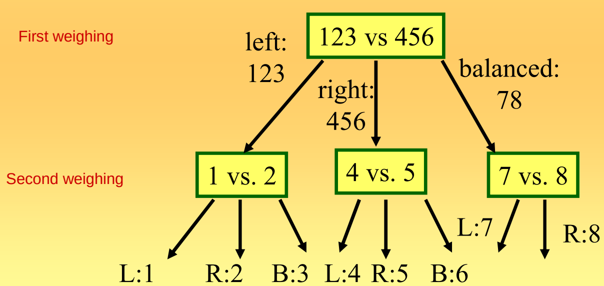
- The problem is an example of searching for 1 unique particular item, from among a list of *n* otherwise identical items.
  - Somewhat analogous to the adage of "searching for a needle in haystack."
- Armed with our balance, we can attack the problem using a divide-and-conquer strategy, like what's done in binary search.
  - We want to narrow down the set of possible locations where the desired item (counterfeit coin) could be found down from *n* to just 1, in a logarithmic fashion.
- **Each weighing has 3 possible outcomes.**
  - Thus, we should use it to partition the search space into 3 pieces that are as close to equal-sized as possible.
- This strategy will lead to the minimum possible worst-case number of weighings required.

General Balance Strategy

- On each step, put $\lceil n/3 \rceil$ of the n coins to be searched on each side of the scale.
  - If the scale tips to the left, then:
    - The lightweight fake is in the right set of $\lceil n/3 \rceil \approx n/3$ coins.
  - If the scale tips to the right, then:
    - The lightweight fake is in the left set of $\lceil n/3 \rceil \approx n/3$ coins.
  - If the scale stays balanced, then:
    - The fake is in the remaining set of $n - 2\lceil n/3 \rceil \approx n/3$ coins that were not weighed!

Coin Balancing Decision Tree

- Here's what the tree looks like in our case: thus 2 weighing are needed only



We are skipping Huffman coding, Prefix coding and Game trees

- Just a word on Huffman
  - It is used whenever data compression is needed.
  - In particular, it is an important part of JEPG code for image compression
  - The basic idea is: if there are n coefficients representing something, **then encode the most frequent coefficient with the shortest bit length**: say A appears 10 times, B-7 times,C-3 times, D-2 times, E-1 time and F-1 time
  - Then A will be encoded by 1, B by 0, C by 10, D by 11, E by 100, and F by 101

10.3 Tree Traversal

- **A traversal algorithm is a procedure for systematically visiting every vertex of an ordered binary tree. Why this may be needed? For example, to perform a task in each node.**
- Traversal algorithms
  - **Preorder** traversal
  - **Inorder** traversal
  - **Postorder** traversal
- Infix/prefix/postfix notation

What Pre-, In- & Post- stand for?

- They tell about
- **where the roots of the tree and subtrees are placed**
- **Pre-** means the roots are the first (**PREcede**)
- **In-** means the roots are **IN the middle**
- **Post-** means the roots are the last (**POST=after**)

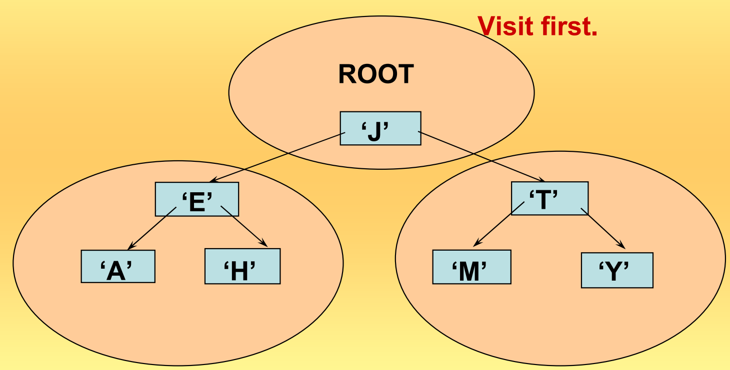
PREORDER Traversal Algorithm

Let T be an ordered binary tree with root r.

If T has only r, then r is the preorder traversal.

Otherwise, suppose T_1, T_2 are the left and right subtrees at r. The preorder traversal begins by visiting r. Then traverses T_1 in preorder, then traverses T_2 in preorder.

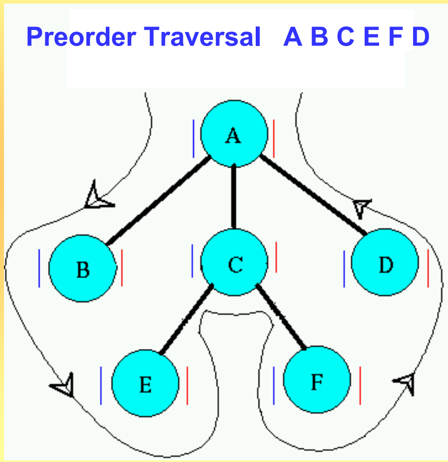
Preorder Traversal J E A H T M Y



Visit left subtree second **Visit right subtree last**

Thus, result, i.e., the Preorder Traversal is: J E A H T M Y

Preorder Traversal



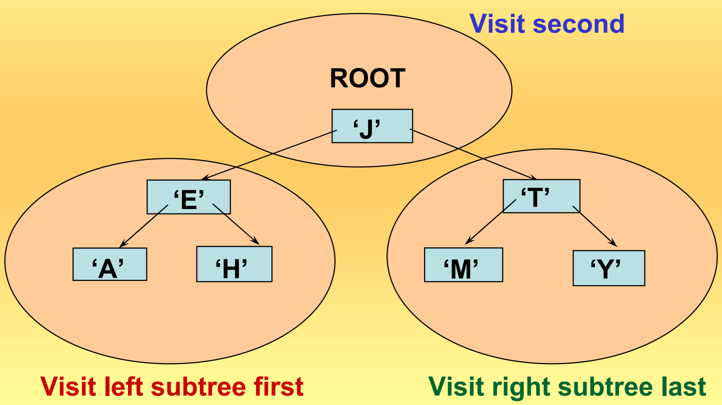
INORDER Traversal Algorithm

Let T be an ordered binary tree with root r.

If T has only r, then r is the inorder traversal.

Otherwise, suppose T_1, T_2 are the left and right subtrees at r. The inorder traversal begins by traversing T_1 in inorder. Then visits r, then traverses T_2 in inorder.

Inorder Traversal A E H J M T Y



Thus, result, i.e., the Preorder Traversal is: A E H J M T Y

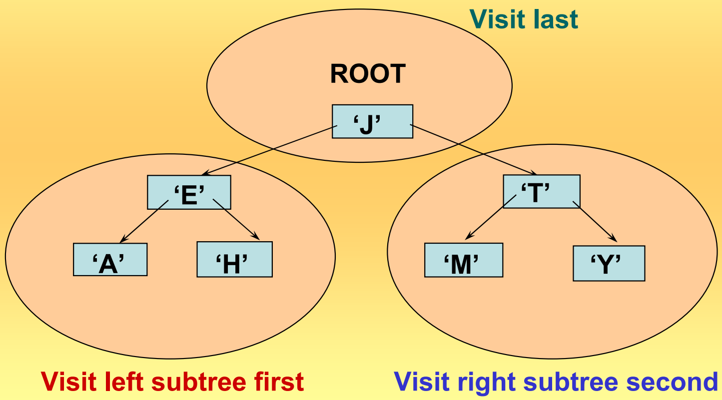
POSTORDER Traversal Algorithm

Let T be an ordered binary tree with root r.

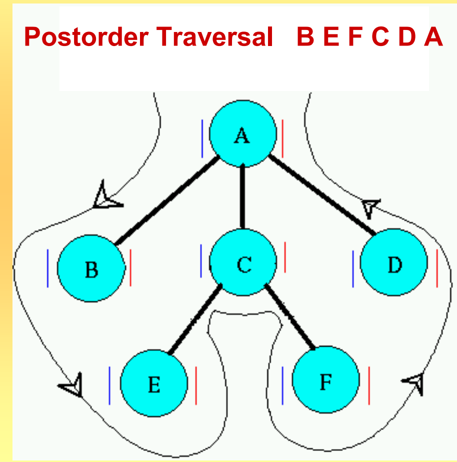
If T has only r, then r is the postorder traversal.

Otherwise, suppose T_1, T_2 are the left and right subtrees at r. The postorder traversal begins by traversing T_1 in postorder. Then traverses T_2 in postorder, then ends by visiting r.

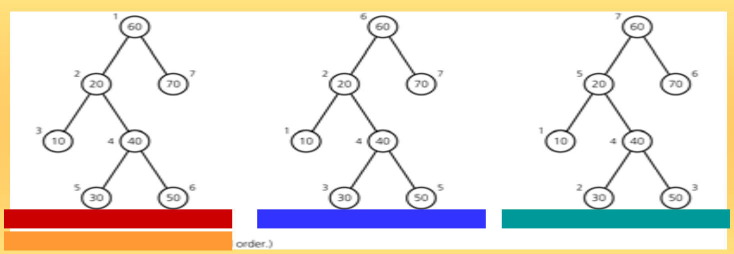
Postorder Traversal A H E M Y T J



Postorder Traversal

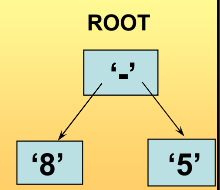


Traversals of a Binary Tree



Traversals of a binary tree: a) preorder; b) inorder; c) postorder

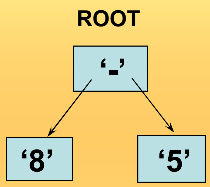
A Binary Expression Tree is . . .



A special kind of binary tree in which:

1. **Each leaf node contains a single operand,**
2. **Each nonleaf node contains a single binary operator, and**
3. **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.**

A Binary Expression Tree

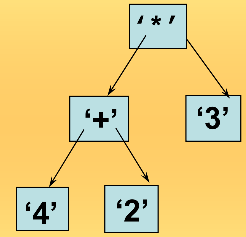


INORDER TRAVERSAL: 8 - 5 has value 3

PREORDER TRAVERSAL: - 8 5

POSTORDER TRAVERSAL: 8 5 -

A Binary Expression Tree



What value does it have?

(4 + 2) * 3 = 18

Levels Indicate Precedence

When a binary expression tree is used to represent an expression, the levels of the nodes in the tree indicate their relative precedence of evaluation.

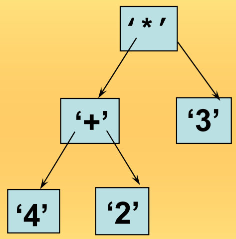
Operations at higher levels of the tree are evaluated later than those below them. The operation at the root is always the last operation performed.

Infix, Prefix, and Postfix Notation are **for your learning.**

Just 2 pages

Next few slides are exercises for these three notations!

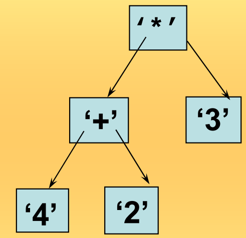
A Binary Expression Tree



What infix, prefix, postfix expressions does it represent?

65/103

A Binary Expression Tree



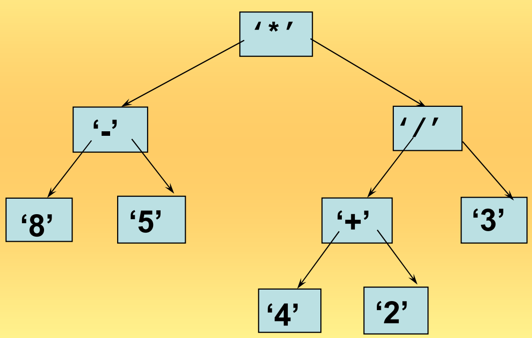
Infix: ((4 + 2) * 3)

Prefix: * + 4 2 3 *evaluate from right*

Postfix: 4 2 + 3 * *evaluate from left*

66/103

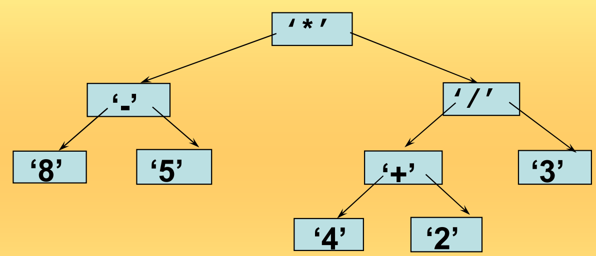
Evaluate this binary expression tree



What infix, prefix, postfix expressions does it represent?

67/103

A binary expression tree



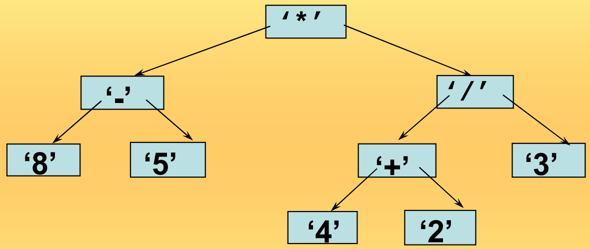
Infix: ((8 - 5) * ((4 + 2) / 3))

Prefix: * - 8 5 / + 4 2 3

Postfix: 8 5 - 4 2 + 3 / * *has operators in order used*

68/103

A binary expression tree

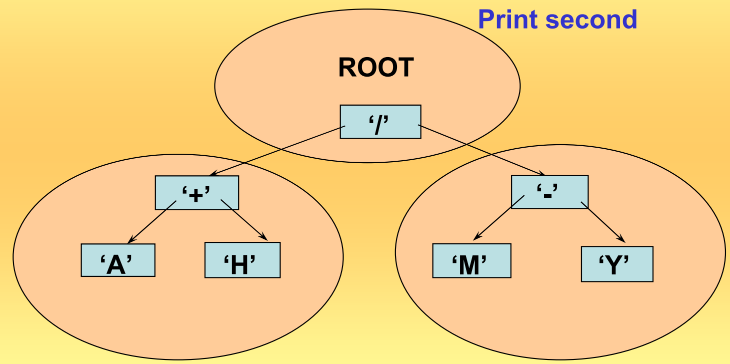


Infix: ((8 - 5) * ((4 + 2) / 3))

Prefix: * - 8 5 / + 4 2 3 *evaluate from right*

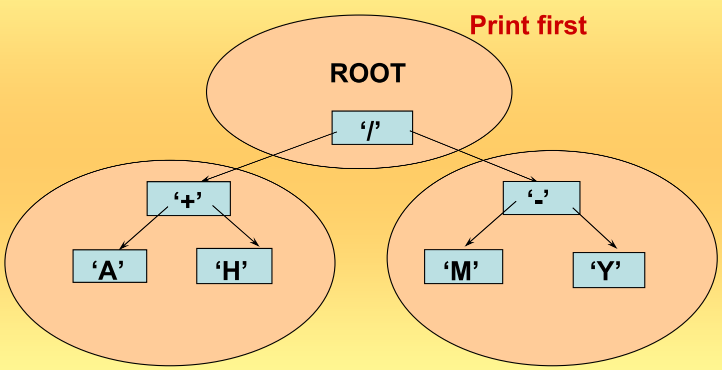
Postfix: 8 5 - 4 2 + 3 / * *evaluate from left*

Inorder Traversal: (A + H) / (M - Y)



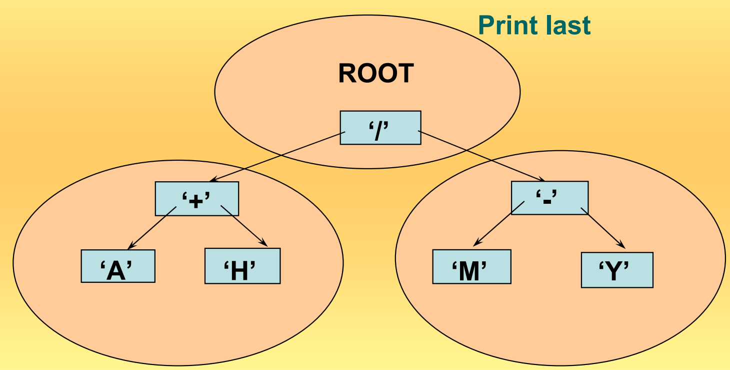
Print left subtree first **Print right subtree last**

Preorder Traversal: / + A H - M Y



Print left subtree second **Print right subtree last**

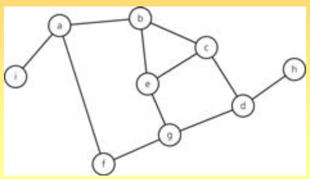
Postorder Traversal: A H + M Y - /

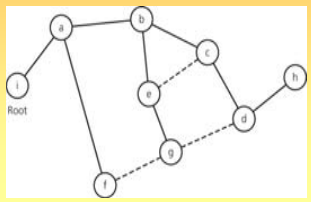


Print left subtree first **Print right subtree second**

10.4 Spanning Trees

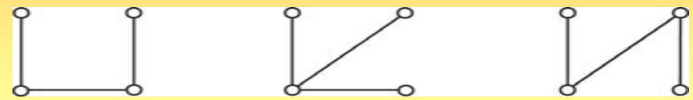
- A tree is an **undirected** connected graph without cycles
- A spanning tree of a connected undirected graph *G is*
  - a subgraph of *G* that contains **all of *G*'s vertices** and enough of its edges to form a tree
- To obtain a spanning tree from a connected undirected graph with cycles
  - Remove edges until there are no cycles





Spanning Trees

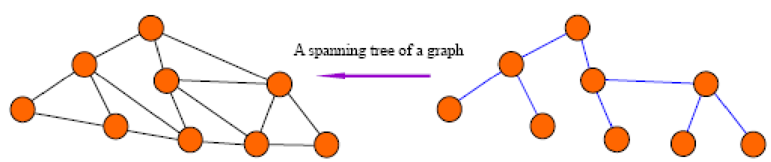
- Detecting a cycle in an **undirected** connected graph
  - A connected **undirected** graph that has n vertices must have at least *n* – 1 edges
  - A connected undirected graph that has n vertices and exactly *n* – 1 edges cannot contain a cycle
  - A connected undirected graph that has n vertices and more than *n* – 1 edges must contain at least one cycle



Connected graphs that each have four vertices and three edges

Spanning trees example

Example



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

How to find a spanning tree of some graph?

Two possibilities:

Depth-First-Search (**DFS**)
Breadth-First-Search (**BFS**)

Run video **Graph Traversals, but first show next slide**

Taken from here: http://www.youtube.com/watch?v=or9xlA3YY

Stack & Queue

- **Stacks**
- A stack is a container of objects that are inserted and removed according to the **last-in first-out (LIFO) principle**. In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. **push** adds an item to the top of the stack, **pop** removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.

- **Queues**
- A queue is a container of objects (a linear collection) that are inserted and removed according to the **first-in first-out (FIFO) principle**. Example: a line of students in the food court at VCU. Additions to a line are made at the back, while removal happens at the front. In the queue only two operations are allowed **enqueue** and **dequeue**. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item.

- The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added

Adopted from Adamchik's lectures, CMU

The DFS Spanning Tree

- **Depth-First Search (DFS)** proceeds along a path from a vertex *v* **as deeply into the graph as possible** before backing up
- To create a depth-first search (DFS) spanning tree
  - Traverse the graph using a depth-first search and mark the edges that you follow
  - After the traversal is complete, the graph's vertices and marked edges form the spanning tree
  - Supporting data structure is a **STACK**

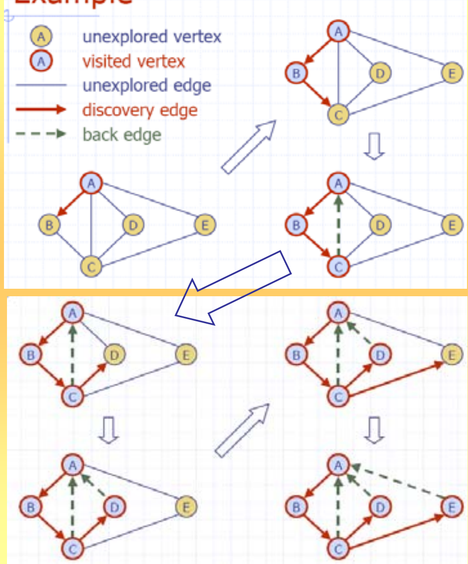
DFS Algorithm for those who like programming and for all to understand how DFS operates

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

```
Algorithm DFS(G)
  Input graph G
  Output labeling of the edges of G
      as discovery edges and
      back edges
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      DFS(G, v)
```

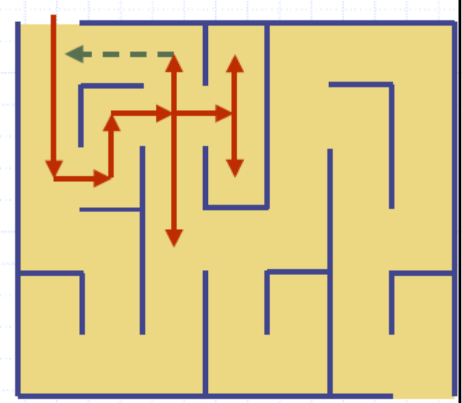
```
Algorithm DFS(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the edges of G
    in the connected component of v
    as discovery edges and back edges
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        DFS(G, w)
      else
        setLabel(e, BACK)
```


Example



The DFS algorithm is similar to a classic strategy for exploring a maze

- We mark each intersection, corner and dead end (vertex) visited
- We mark each corridor (edge) traversed
- We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



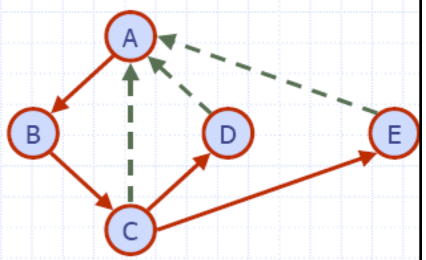
Properties of DFS

Property 1

$DFS(G, v)$ visits all the vertices and edges in the connected component of v

Property 2

The discovery edges labeled by $DFS(G, v)$ form a spanning tree of the connected component of v



Analysis of DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$

Path finding

- We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern
- We call $DFS(G, u)$ with u as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, we return the path as the contents of the stack

```
Algorithm pathDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  if v = z
    return S.elements()
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        S.push(e)
        pathDFS(G, w, z)
        S.pop(e)
      else
        setLabel(e, BACK)
  S.pop(v)
```


Cycle finding

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

```
Algorithm cycleDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      S.push(e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        pathDFS(G, w, z)
        S.pop(e)
      else
        T ← new empty stack
        repeat
          o ← S.pop()
          T.push(o)
        until o = w
        return T.elements()
  S.pop(v)
```

The BFS Spanning Tree

- Breadth-First Search (BFS) visits every vertex adjacent to a vertex *v* that it can before visiting any other vertex
- To create a breath-first search (BFS) spanning tree
  - Traverse the graph using a bread-first search and mark the edges that you follow
  - When the traversal is complete, the graph's vertices and marked edges form the spanning tree
  - Supporting data structure is a **QUEUE**

BFS basics

- Breadth first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- BFS on a graph with n vertices and m edges takes $O(n+m)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

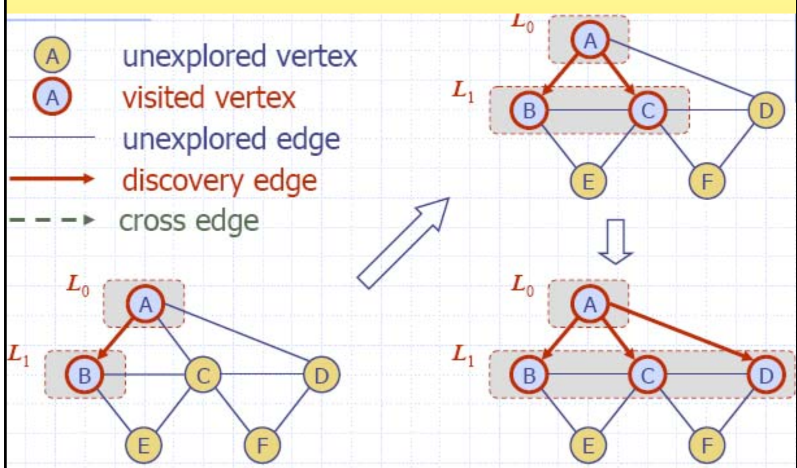
BFS algorithm

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

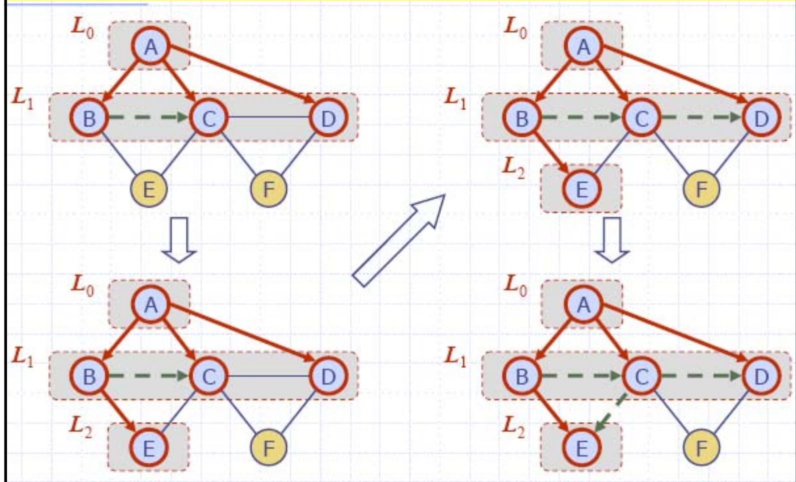
```
Algorithm BFS(G)
  Input graph G
  Output labeling of the edges
     and partition of the
     vertices of G
 for all u ∈ G.vertices()
  setLabel(u, UNEXPLORED)
 for all e ∈ G.edges()
  setLabel(e, UNEXPLORED)
 for all v ∈ G.vertices()
  if getLabel(v) = UNEXPLORED
     BFS(G, v)
```

```
Algorithm BFS(G, s)
  L0 ← new empty sequence
  L0.insertLast(s)
  setLabel(s, VISITED)
  i ← 0
  while ¬Li.isEmpty()
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            Li+1.insertLast(w)
          else
            setLabel(e, CROSS)
    i ← i +1
```

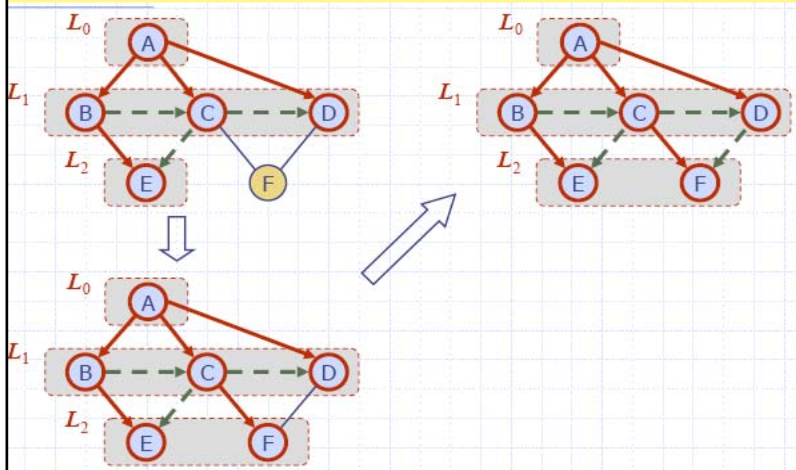

BFS - example



BFS-example cont.



BFS-example cont. 2



BFS properties

Notation

G_s: connected component of s

Property 1

$BFS(G, s)$ visits all the vertices and edges of G_s

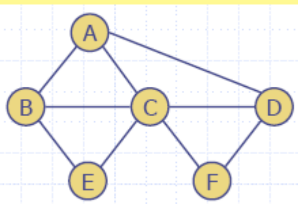
Property 2

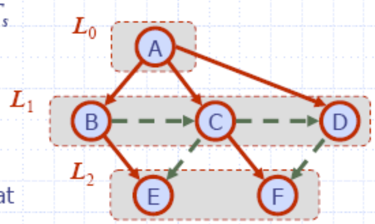
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

- The path of T_s from s to v has i edges
- Every path from s to v in G_s has at least i edges





BFS analysis

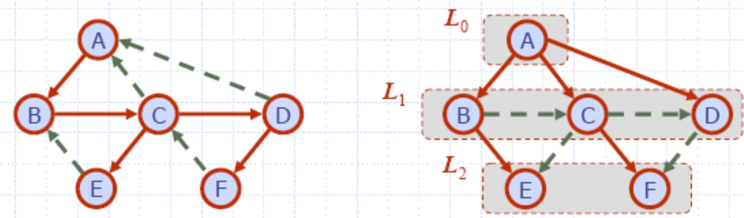
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$

BFS applications

- Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	

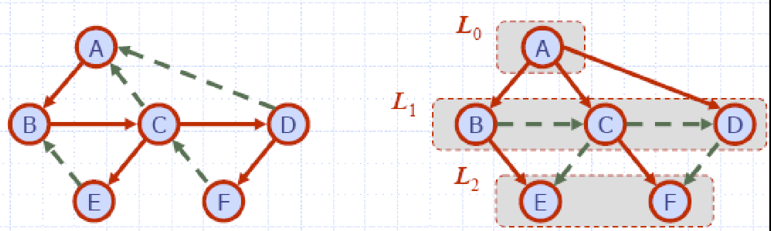


DFS BFS

DFS vs. BFS

Back edge (v,w)
- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)
- w is in the same level as v or in the next level in the tree of discovery edges



DFS BFS

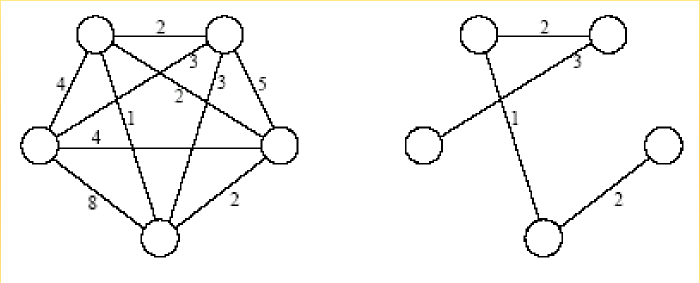
10.5 Minimum Spanning Trees

- A *spanning tree* of an **undirected** graph *G* is a subgraph of *G* that is a tree containing all the vertices of *G*.
- In a **weighted graph**, the weight of a subgraph is the sum of the weights of the edges in the subgraph.
- A ***minimum spanning tree*** (MST) for a **weighted undirected** graph is a spanning tree with minimum weight.
-
- Note: Weight can be anything – length of the link, time needed to pass the link, cost of the link etc,…

Minimum Spanning Trees

- Cost of the spanning tree
  - Sum of the costs of the edges of the spanning tree
- A minimal spanning tree of a connected **undirected** graph has a minimal edge-weight sum
  - There **may be several minimum spanning** trees for a particular graph

MST - example



An **undirected** graph and its minimum spanning tree.

Prim's Algorithm - idea

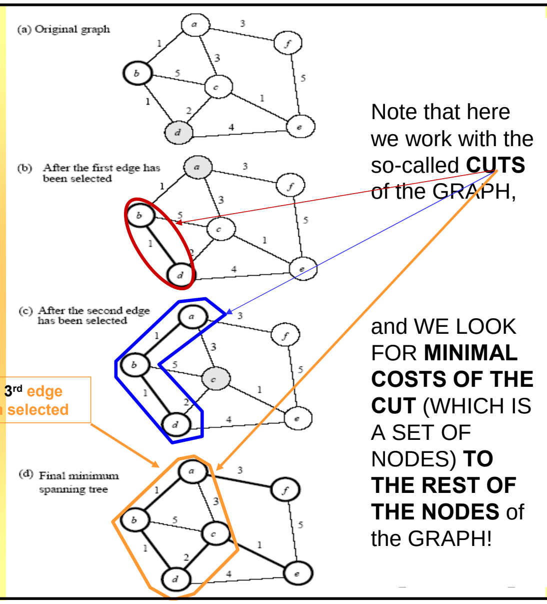
The algorithm was invented in **1930 by Czech mathematician Vojtech Jarník,** and reinvented by Prim in 1957, as well as by Dijkstra in 1959!

- Prim's algorithm for finding an MST is a **greedy* algorithm.**
- Start by selecting an **arbitrary** vertex, include it into the current MST.
- Grow the current MST by inserting into it the vertex closest **to one of** the **vertices already in current MST**.

*A **greedy algorithm** is any algorithm that follows the problem solving metaheuristic of making **the locally optimal choice** at each stage with the hope of finding the global optimum. Note, however, that the **sum of local optima is not necessarily a global optimum!**

Prim's Algorithm

- finds a minimal spanning tree that begins generally **at any**, or **at a given**, vertex
  - Find the least-cost edge (*v*, *u*) from a visited vertex *v* to some unvisited vertex *u*
  - Mark *u* as visited
  - Add the vertex *u* and the edge (*v*, *u*) to the minimum spanning tree
  - Repeat the above steps until there are no more unvisited vertices



After the 3rd edge has been selected

Note that here we work with the so-called **CUTS** of the GRAPH,

and WE LOOK FOR **MINIMAL COSTS OF THE CUT** (WHICH IS A SET OF NODES) **TO THE REST OF THE NODES** of the GRAPH!

In your textbook there is one more algorithm for finding MST named after KRUSKAL.

- It works differently than Prim's one, but it is a greedy algorithm too.
- Check your book about the details, if interested.

This ends the Trees Story here!!!

And now, back to Chapter 4 on Induction and Recursion