Department of Computer Science
National Tsing Hua University

CS 2336: Discrete Mathematics

Chapter 12

Trees

Instructor: Cheng-Hsin Hsu

Outline

Tree

- Consider a loop-free undirected graph G=(V,E). It is a tree if G is connected and contains no cycles
- We often refer to a tree as T instead of (more general) G
- Spanning tree: a spanning subgraph that is also a tree
- Spanning forest: a unconnected spanning subgraph

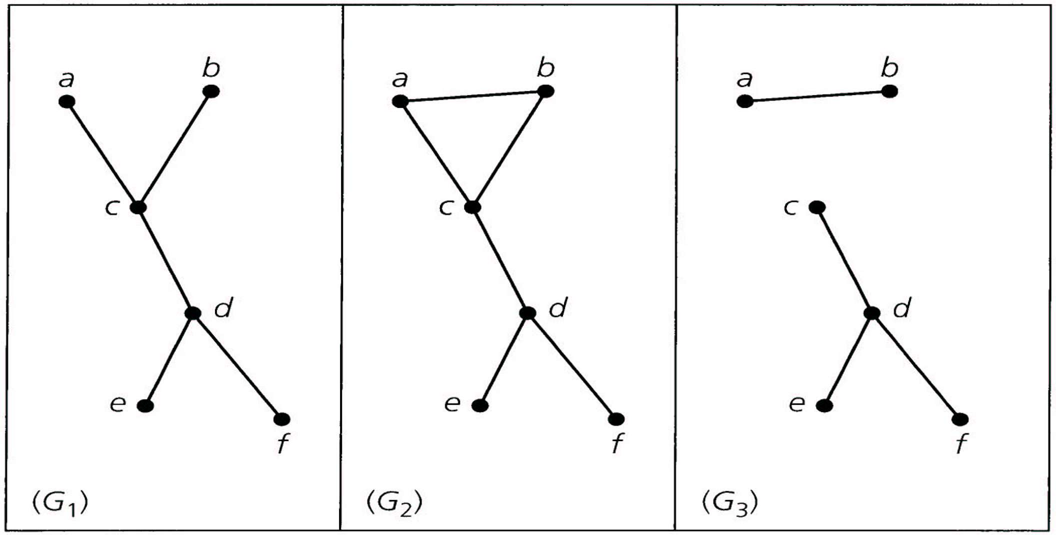


Figure 12.1

Properties of Trees

- Unique path: there exists a unique path between any two distinct vertices in T=(V,E)
  - Proof Sketch: T is connected, so there must be at least one path. Moreover, if there are two paths, connecting them gives us a cycle.
- If G=(V,E) is an undirected graph, G is connected iff G has a spanning tree
  - Proof Sketch: (←) by G is connected. (→) Build a spanning tree by iteratively removing an edge on any cycle.

Relation between |V| and |E|

- Counts |V| and |E| in these trees

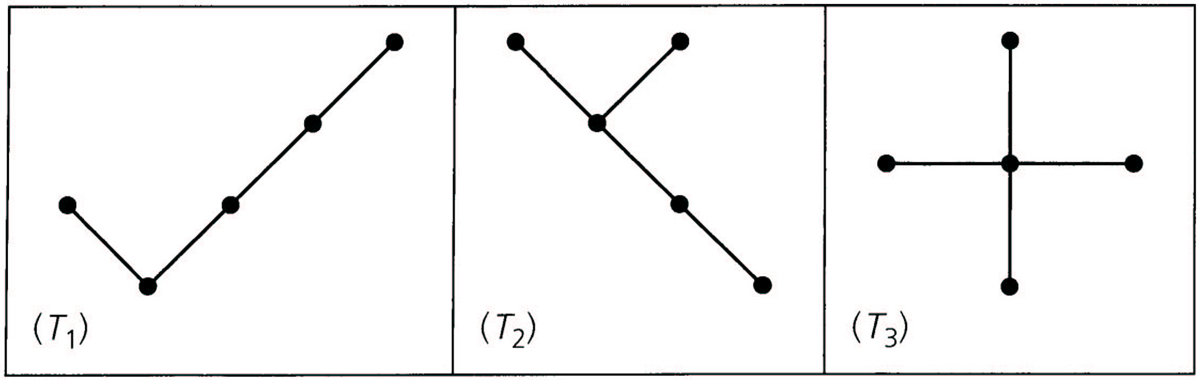


Figure 12.2

- In any tree T=(V,E), we have |V| = |E|+1
  - Proof Sketch: by mathematical induction

Pendant Vertices

- Counts no. pendant vertices in these trees

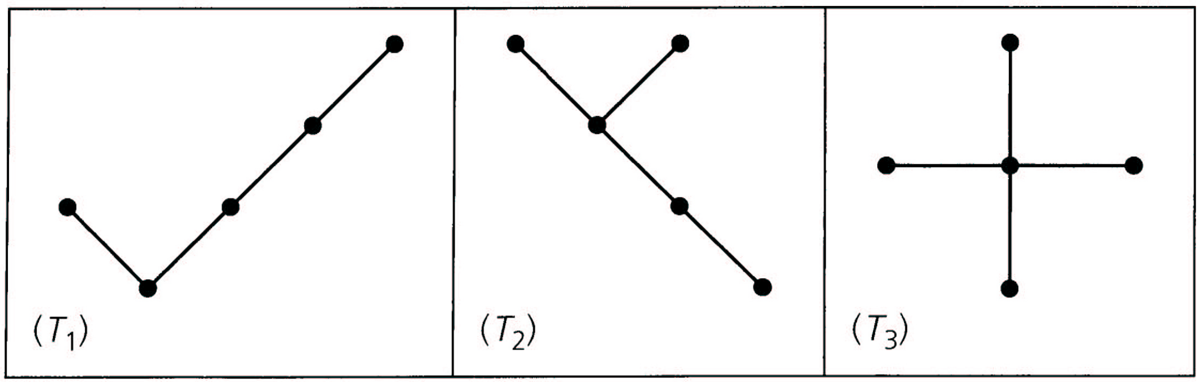


Figure 12.2

- In any tree T=(V,E), where |V| >= 2, T has at least two pendant vertices
  - Proof Sketch: by the previous theorem and $2|E| = \sum_{v \in V} deg(v)$

Examples

- Ex 12.1: Are the two trees isomorphic? Why?

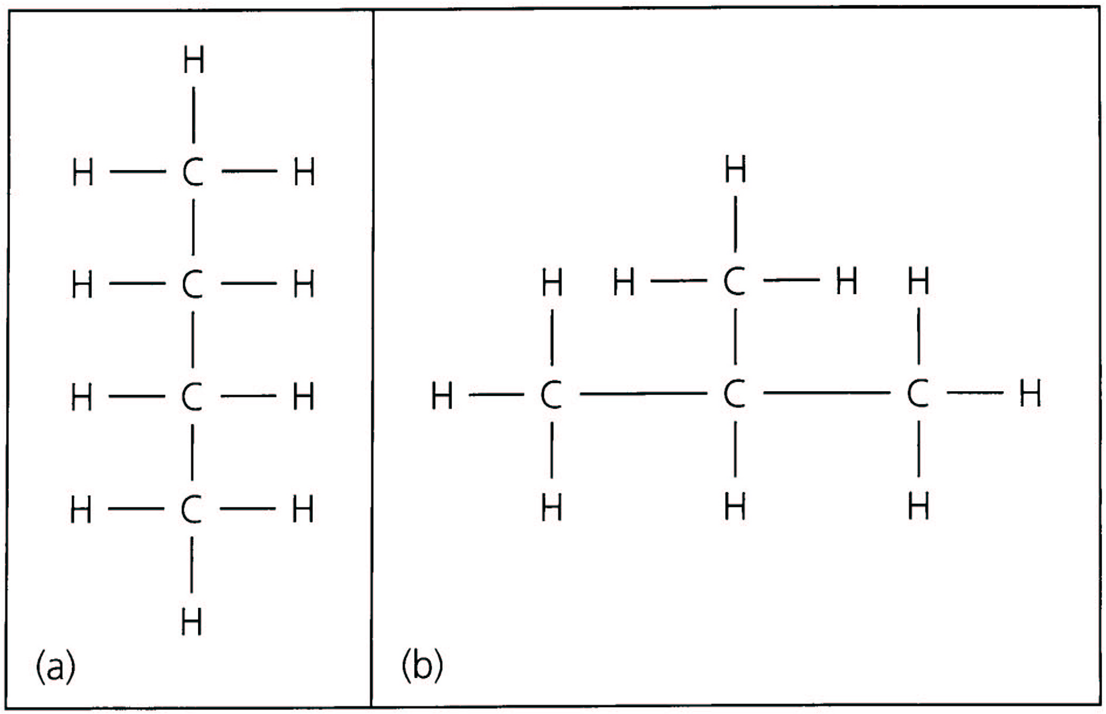


Figure 12.5

Examples

- Ex 12.2: If a saturated hydrocarbon (acyclic) has n carbon atoms, show that it has $2n+2$ hydrogen atoms.
- Proof:
  - Let k denote the number of hydrogen atoms. The total degree of all atoms is $4n+k$, which equals to $2|E|$
  - We also know $|E|=|V|-1$, so the total degree$=2|V|-1$
  - This leads to $k = 2n+2$

When Can We Call a Graph Tree?

- The following statements are equivalent for a look-free undirected graph G=(V,E)
  - G is a tree
  - G is connected, but remove any edge from G turns G into two trees
  - G contains no cycles, and |V|=|E|+1
  - G is connected, and |V|=|E|+1
  - G contains no cycle and if {a,b} is not an edge of G, adding {a,b} to G results in exactly one cycle

A Sample Proof

- Prove if
  - G is a tree,
  - then G is connected, but remove any edge from G turns G into two trees
- Proof:
  - Let G'=G-{a,b}. Assume G' is still connected, which means there is a path between a and b. But this contradict to the fact that tree is acyclic. Hence, G' is not connected!
  - Then consider the two components in G', they must contain no cycles (otherwise G is not a tree). Then they are both trees. This yield our proof.
- See text and exercises for more proofs.

Outline

12.1 Definitions, Properties, and Examples

12.2 Rooted Trees

12.3 Trees and Sorting

12.4 Weighted Trees and Prefix Codes

12.5 Biconnected Components and Articulation Points

Directed and Rooted Trees

- If G is a directed graph, G is a directed tree if its associated undirected graph is a tree
- A directed tree is a rooted tree, if there is a unique vertex r with in-degree 0, id(r)=0, while all other vertex v has in-degree 1, id(v)=1. We call this v as the root.

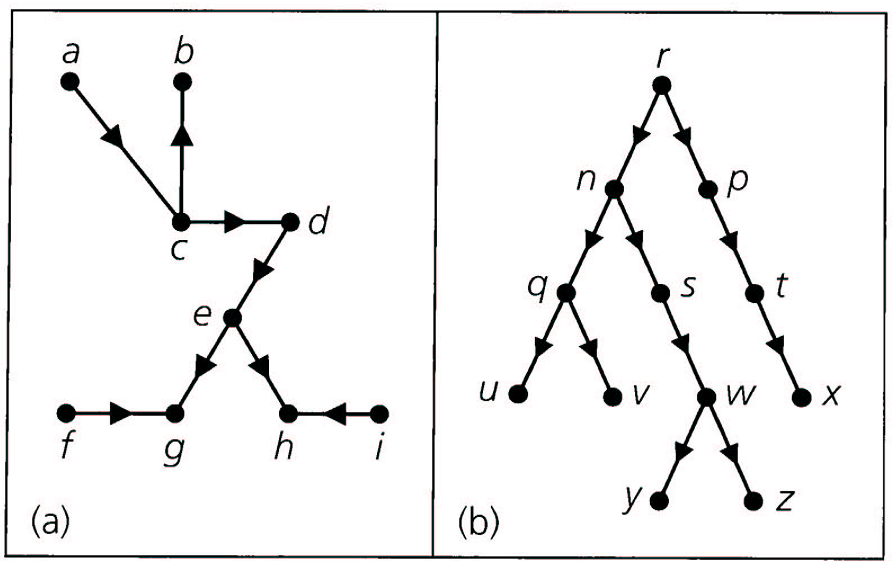


Figure 12.10

Conventions and Terminology

- Arrows are going downwards
- Vertices with zero out degree are call leaves (terminal vertices)
- All other leaves are called branch nodes (or internal vertices)
- Level is defined as the distance to the root
- Parent-child relation, Ancestors-descendants ,Siblings
- Subtree, induced by a vertex v, includes v and all its descendants

Vertex Ordering

- Ex 12.3: Consider a book with 3-level structure. What is the nature order of its contents?

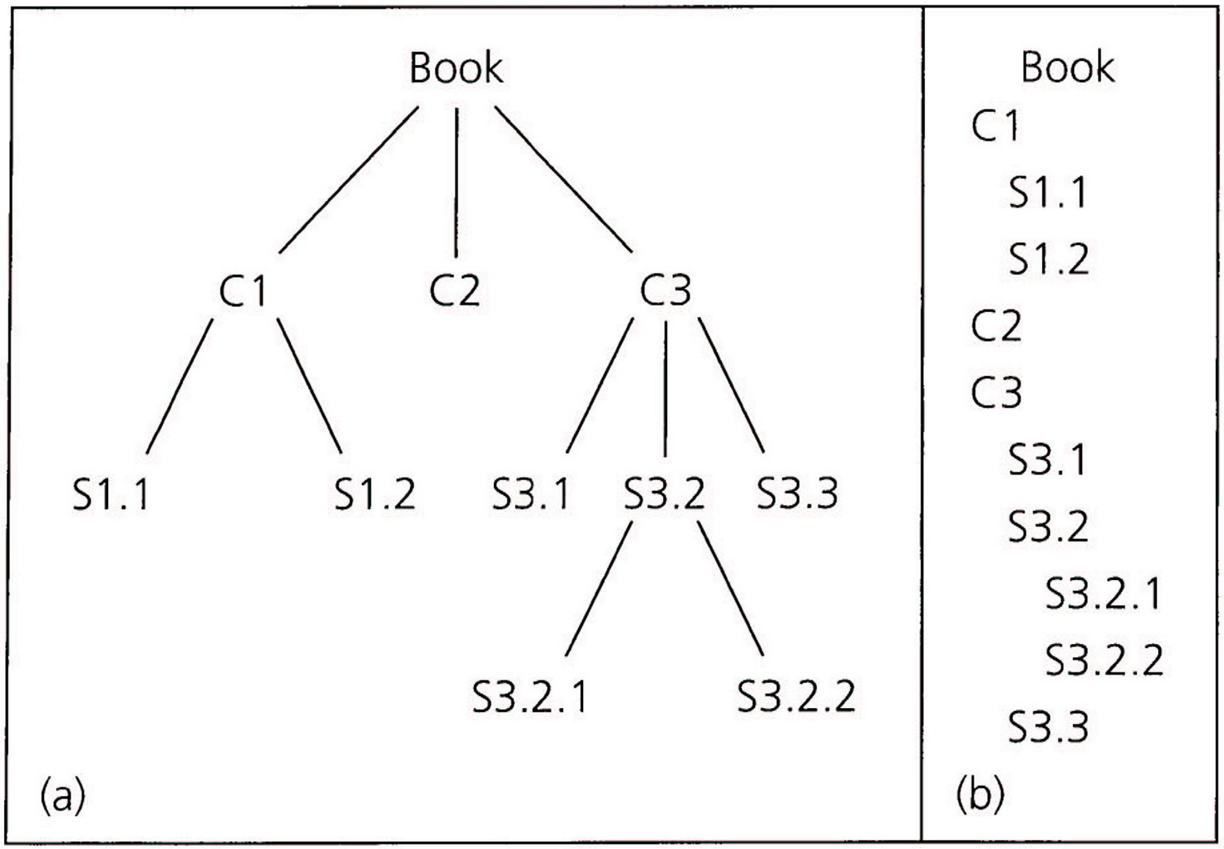


Figure 12.11

Ordered Rooted Tree

- Ex 12.4: If all edges leaving an internal vertex are ordered from left to tight, then T is called an ordered rooted tree.
- Ordering algorithm
  - Assign 0 to the root
  - Assign positive integer to vertices at level 1, from left to right
  - For an internal vertex v, suffix a positive integer to v's label, from left to right

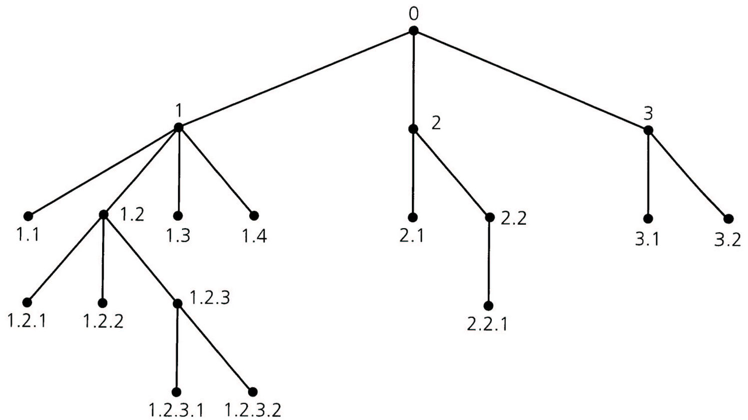


Figure 12.12

Ordered Rooted Tree (cont.)

- This leads to the order:
  - 0, 1, 1.1
  - 1.2, 1.2.1, 1.2.2
  - 1.2.3, 1.2.3.1, 1.2.3.2
  - 1.3, 1.4, 2
  - 2.1, 2.2, 2.2.1
  - 3, 3.1, 3.2
- Lexicographic order

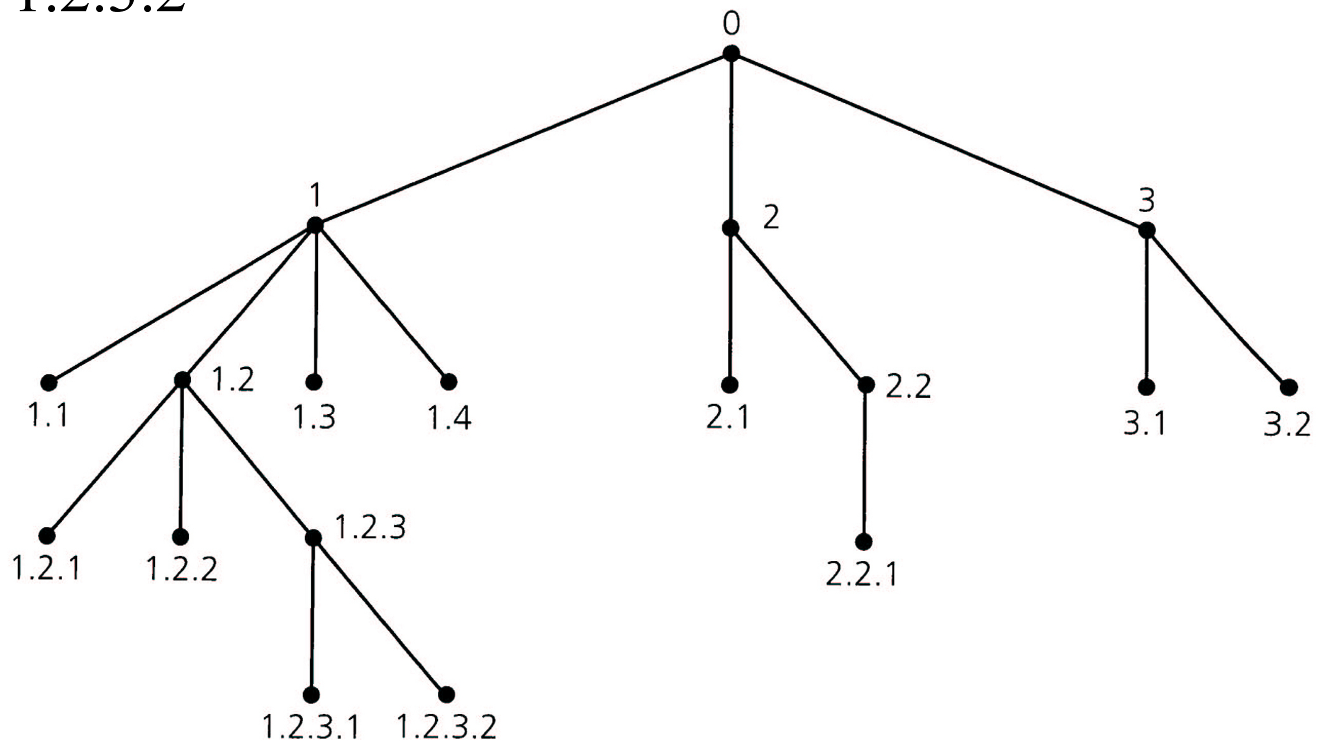


Figure 12.12

Binary Rooted Tree

- Ex 12.5: Binary rooted tree: od(v)=0,1,2. Complete binary tree: od(v)=0,2
- They can represent binary operations

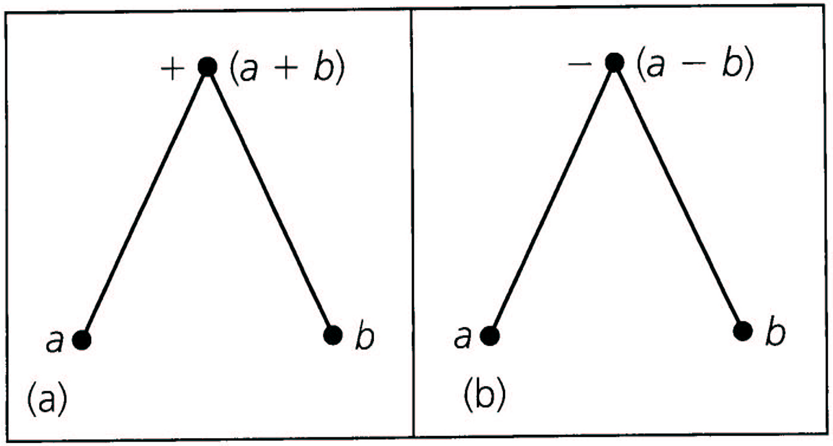


Figure 12.13

Binary Rooted Tree (cont.)

- A tree for ((7-a)/5)*((a+b)^3)

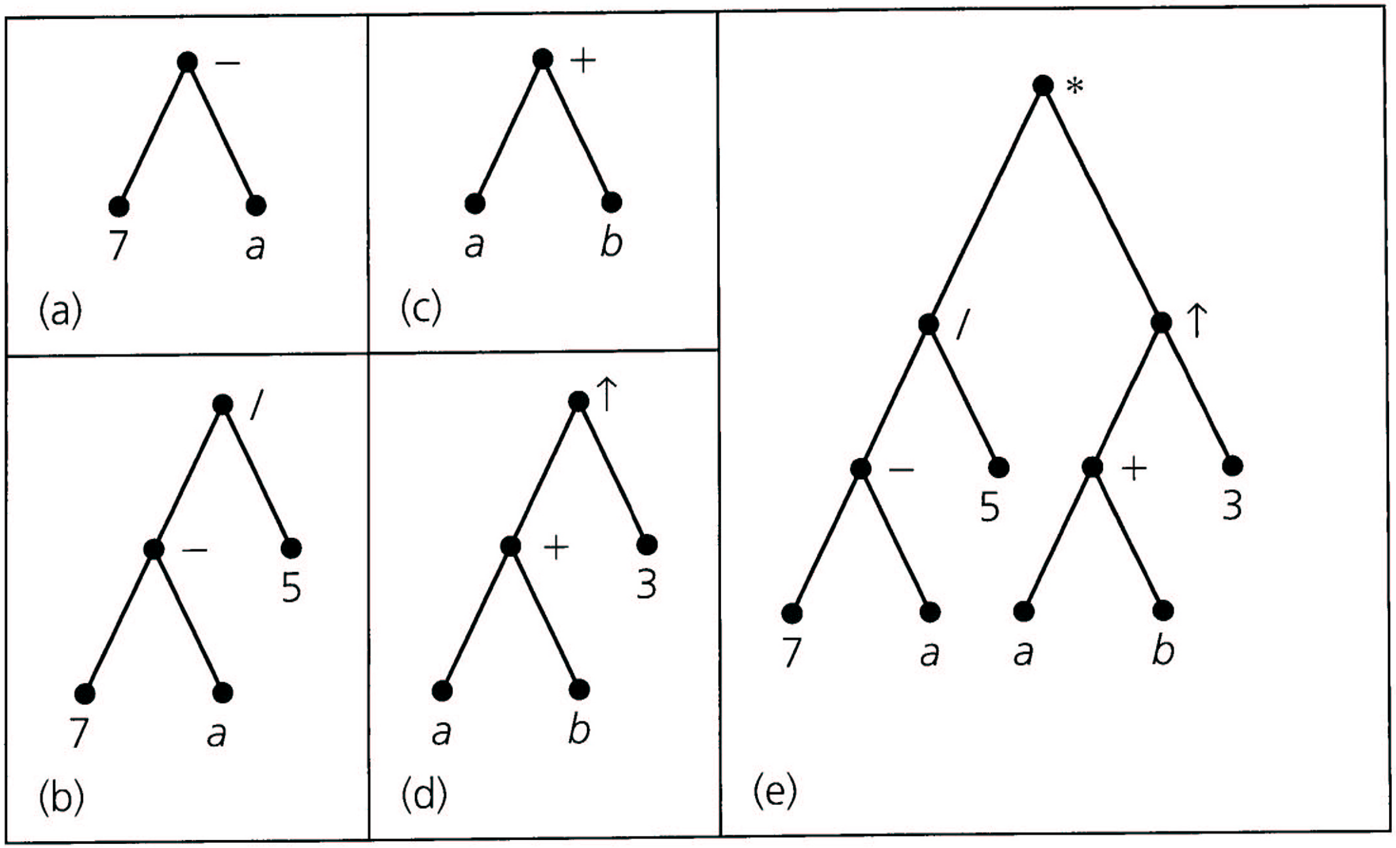


Figure 12.14

Binary Rooted Tree (cont.)

- How to represent: (i) (a-(3/b))+5 and (ii) a-(3/(b+5))
- Both of them can be stored as the same sequence
- Parenthesis are mandatory!

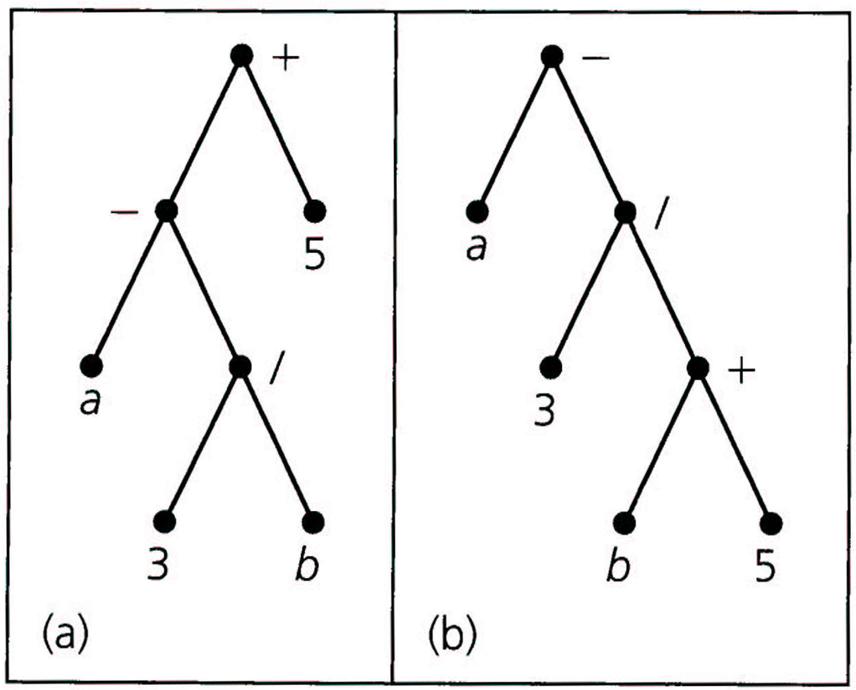


Figure 12.15

Polish Notation

- Consider t+(uv)/(w+x-y^z), it can be expressed by

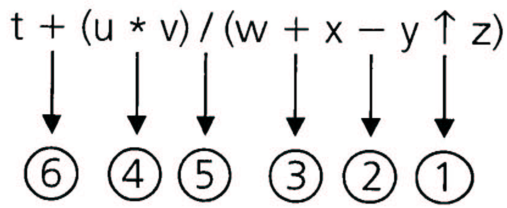


Figure 12.16

- The computer needs to know the calculation order ← But the computer needs to know the parenthesis

- Prefix notation: +t/*uv+w-x^yz

- Independent to parenthesis! Just calculate from right to left ← shows the importance of ordering

Polish Notation (cont.)

- Example:
  - + 4/*23+1-9^23
  - +4/*23+1-98
  - +4/*23+11
  - +4/*232
  - +4/62
  - +43
  - 7

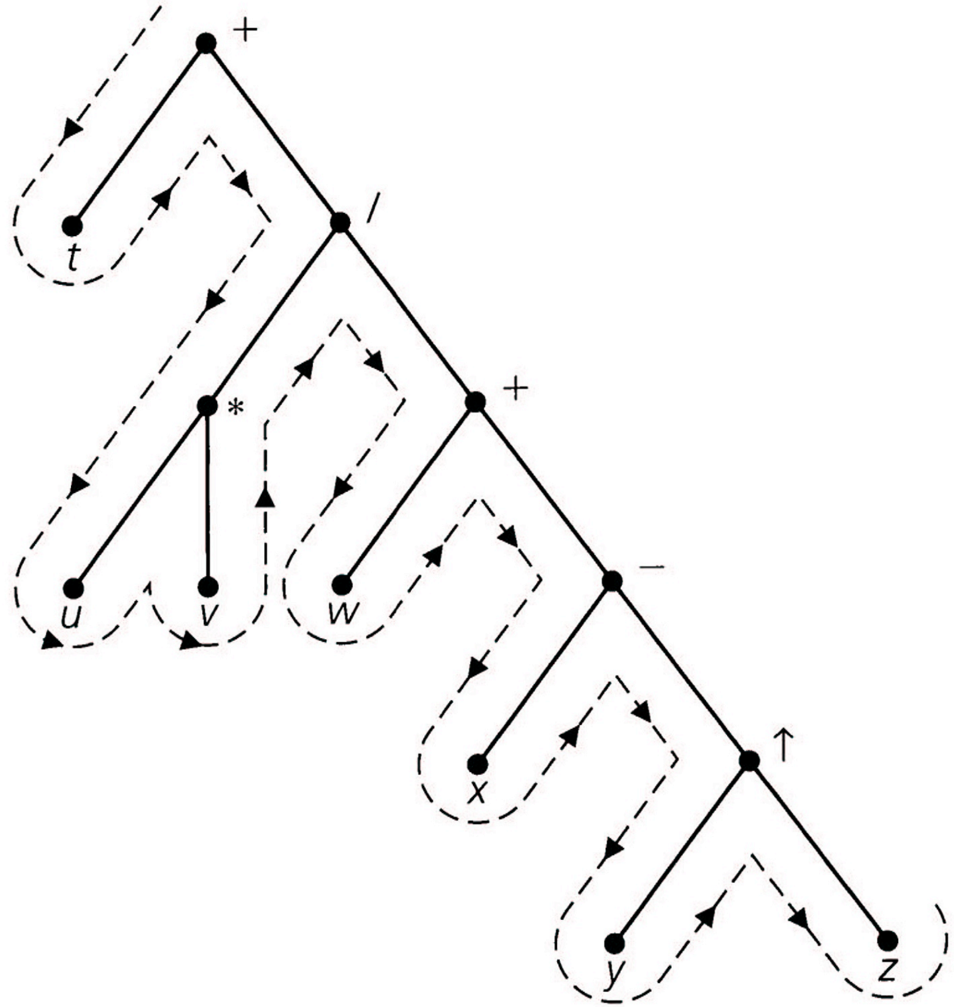


Figure 12.17

Post-/Pre-order Traversals

- Recursively defined
- Let T=(V,E) be a rooted tree with root r
  - If |V|=1, then r is both postorder and preorder traversal
  - Otherwise, preorder traversal first visits r and then traverse subtrees T_1, T_2, …., T_k. Postorder traversal first visits subtrees , then r
  - Conventionally, subtrees are visited from left to right

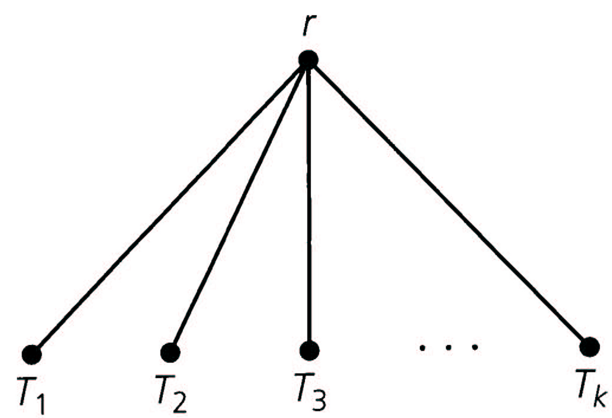


Figure 12.18

Example

- Ex 12.6: What are the pre-/post-order traversals of this graph?

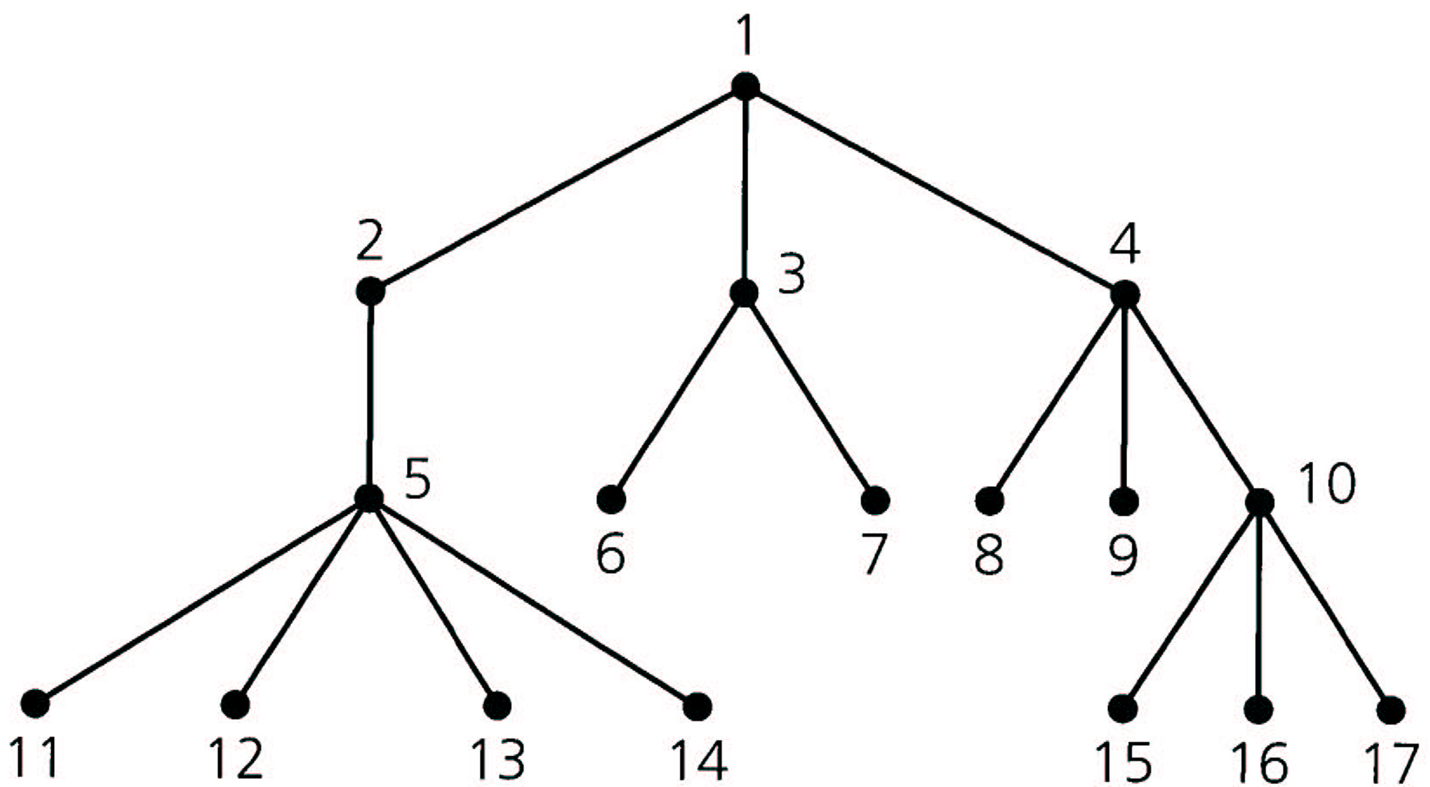


Figure 12.19

In-order Traversal

- For binary rooted tree, we also have in-order traversal
- Let T=(V,E) be a binary rooted tree with root r
  - If |V|=1, then r is the inorder traversal
  - Otherwise, let TL and TR be the left and right subtrees. The inorder traversal first traverses TL, then visits r, and then traverses TR.

Different Ordering

- Ex 12.7:
  - The following two ordered trees are different
  - What are their inorder traversals?
  - What are their preorder and postorder traversals?

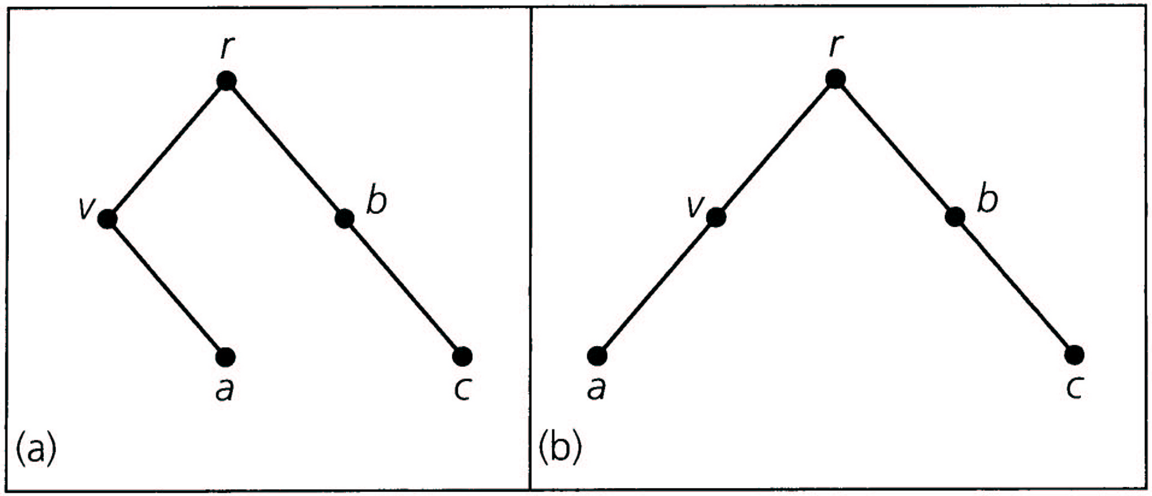


Figure 12.20

Another Inorder Example

- What is the in order traversal?

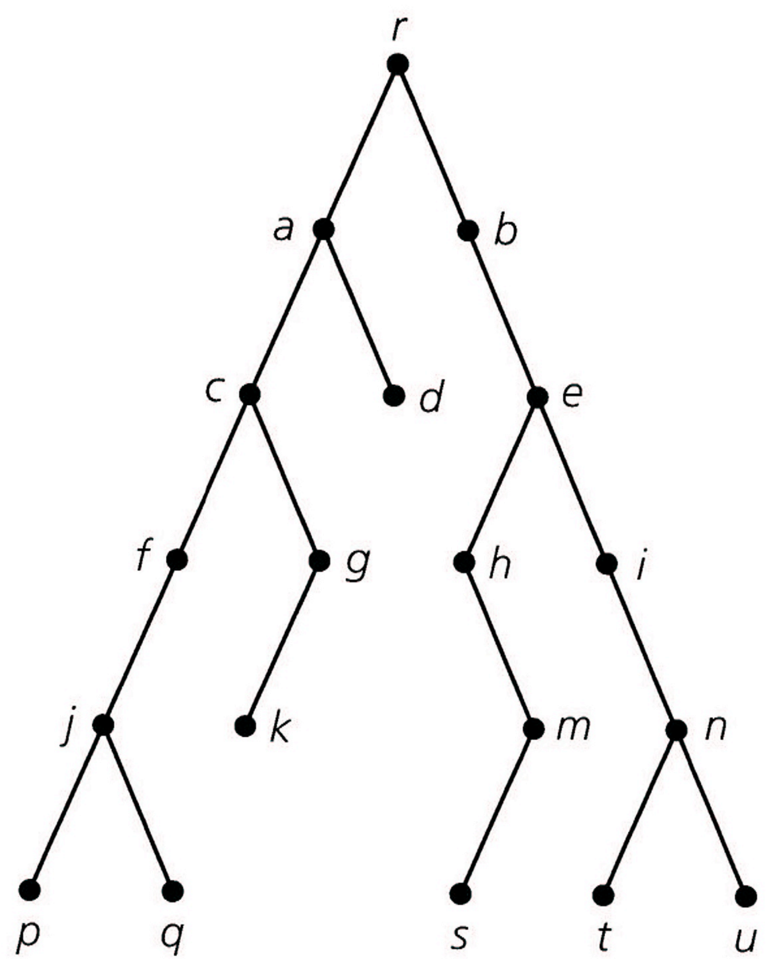
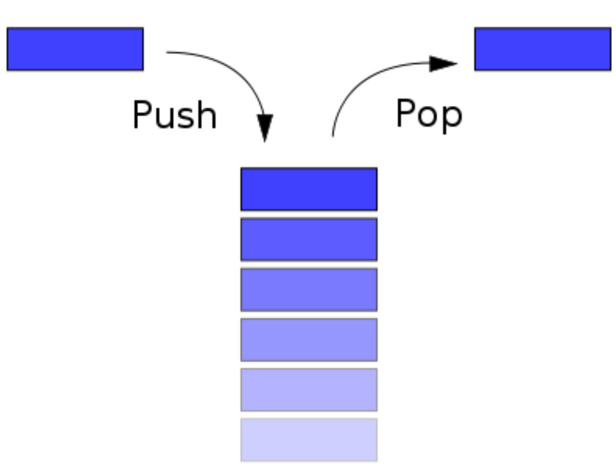
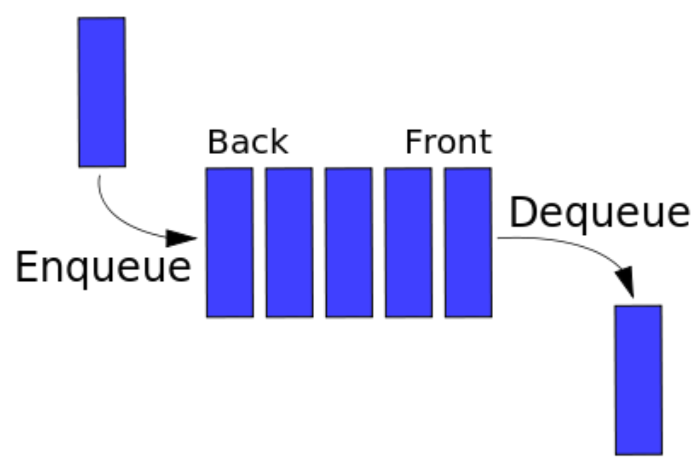


Figure 12.21

Spanning Trees

- Generally two algorithms to generate a spanning trees in a graphs
- Depth-First Search (DFS): based on a stack
- Breadth-First Search (BFS): based on a (FIFO) queue





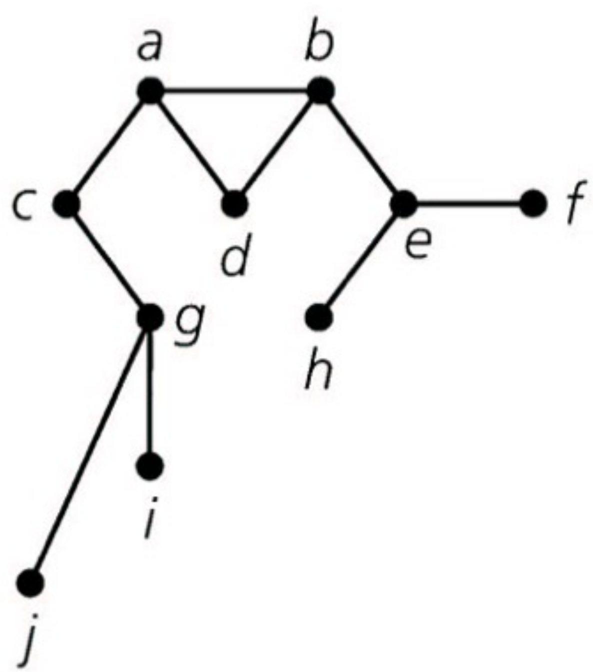
Figures are from Wikipedia

DFS Algorithm

- Let $v=v_1$ as the root of tree T
- If G has only one vertex, terminates and return T
- Select the smallest subscript i, so that $\{v,v_i\}$ is an edge of G and v_i hasn't been visited
- If an i exists: (i) add $\{v,v_i\}$ to T, (ii) visit subtree induced by v_i, (iii) let $v=v_i$, go back to the step 3
- If there is no v_i, then backtrack from v to its parent u. Let v = u, and go back to step3
- Once all vertices are visited, return T

Example of DFS

- Ex 12.10: Plot the DFS trees of graph G
  - Assuming the vertex order is: a,b,c,d,e,f,g,h,i,j
  - Assuming the order is: j,i,h,g,f,e,d,c,b,a



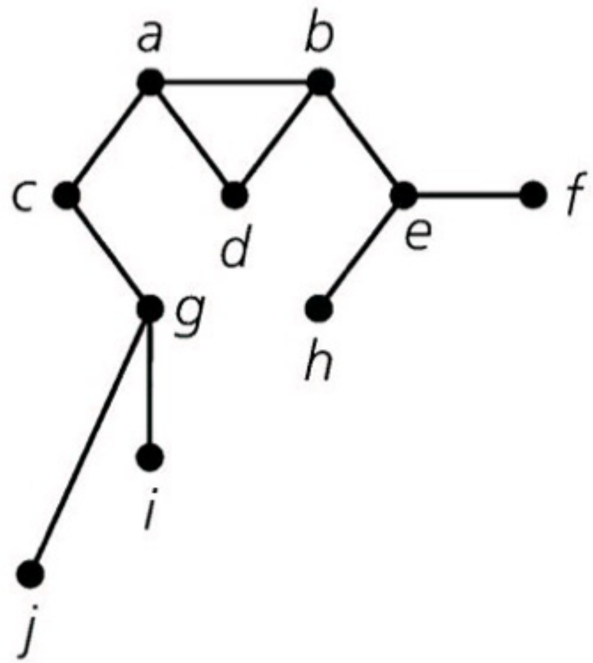
(a) $G = (V, E)$

BFS Algorithm

- Enqueue v_1, and let T be the tree with v_1, visit v_1
- Let v=dequeue(). Sequentially check all vertices next to v that haven't been visited
- For each unvisited vertex v_i: (i) enqueue v_i, (ii) add $\{v, v_i\}$ to T, and (iii) visit v_i
- If queue is not empty go to step 2
- Now queue is empty, return T

Example of BFS

- Ex 12.11: Plot the BFS trees of graph G
  - Assuming the vertex order is: a,b,c,d,e,f,g,h,i,j
  - Assuming the order is: j,i,h,g,f,e,d,c,b,a



(a) $G = (V, E)$

Adjacent Matrix to BFS/DFS Trees

- Ex 12.12 Determine the BFS and DFS tress from the adjacent matrix without plotting the graph

$$A(G) = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{array} \begin{array}{c} \begin{array}{ccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

M-ary Tree

- Let T=(V,E) be a rooted tree, and m is a positive integer. T is called an m-ary tree if od(v)<=m for all v
- When m=2, it is called a binary tree
- If od(v)=0 or m, for all v, then T is called a complete m-ary tree.
  - Each internal vertex has m children
- When m=2, it is called a complete binary tree.

Property of a Complete m-ary Tree

- Let T=(V,E) be a complete m-ary tree with |V|=n. If T has l leaves and i internal vertices then
  - $n=mi+1$ ← each internal node leads to m children, plus root
  - $l=(m-1)i+1$ ← based on equation 1 and n=l+i
  - $i=(l-1)/(m-1)=(n-1)/m$ ← base don equations 1 and 2

Number of Matches

- Ex 12.13: In a single-elimination tournament. If there are 27 players, how many matches must be played to determine the champion?
  - 27 players, so 27 leaves (l=27), also m=2. Therefore, we have i=(l-1)/(m-1)=(27-1)/(2-1)=26

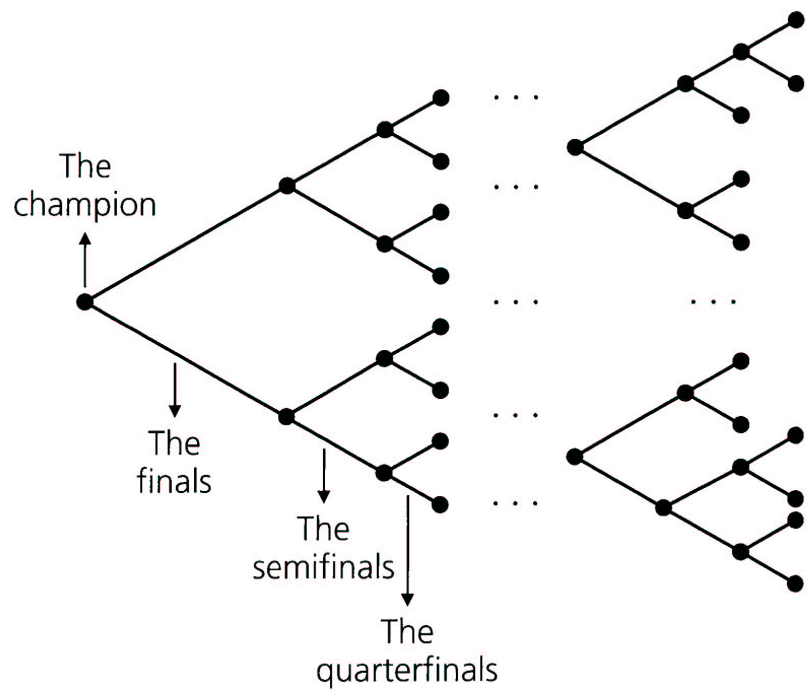


Figure 12.26

Height and Balanced Trees

- Let T=(V,E) be a rooted tree, and h be the largest level number by a leaf of T. We say T has a height of h.
- A rooted tree T of height h is balanced if the level number of every leaf is either h or h-1.

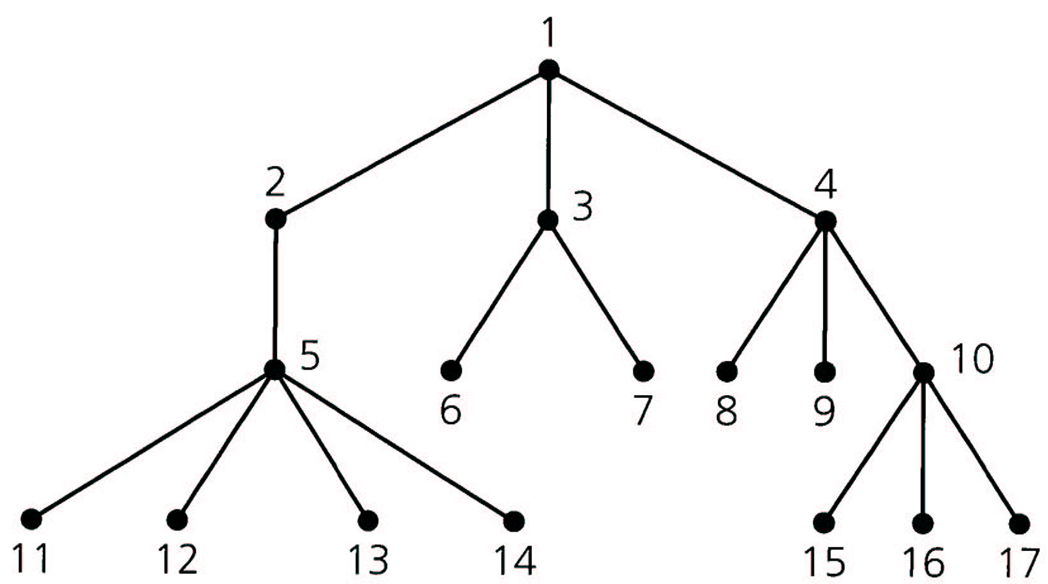


Figure 12.19

Height of m-ary Tree

- Let T=(V,E) be a complete m-ary tree with height *h* and *l* leaves. We have $l \leqslant m^h$ and $h \geqslant \lceil \log_m l \rceil$
  - Proved by induction

- Let T be a balanced complete m-ary tree with *l* leaves. The height of T is $\lceil \log_m l \rceil$

Decision Tree

- There are 8 coins and a pan balance. One of the coin is counterfeit and heavier than others. Find that coin.
- Binary decision tree $h \geqslant \lceil \log_2 8 \rceil$
- Ternary decision tree $h \geqslant \lceil \log_3 8 \rceil$

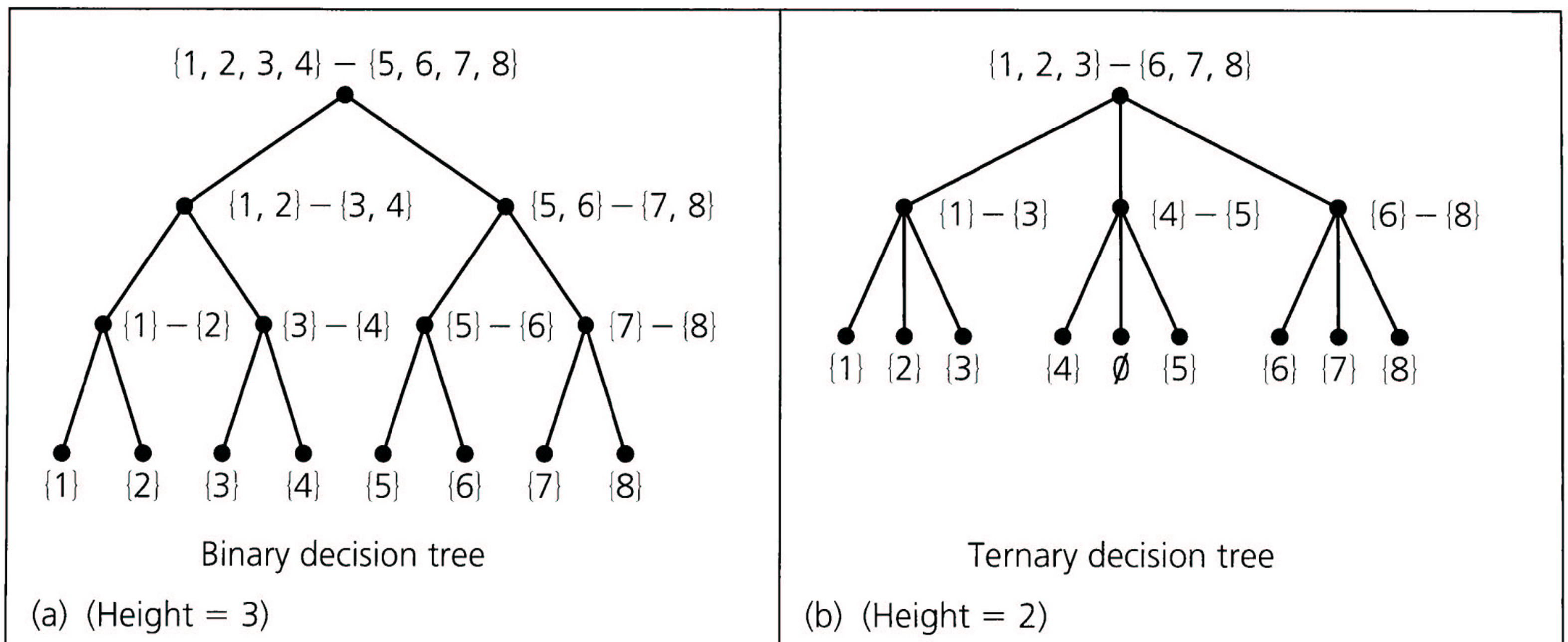


Figure 12.27

Outline

Bubble Sort

- Simplest sorting algorithm
- High complexity: $O(n^2)$

```
procedure BubbleSort(n: positive integer; x_1, x_2, x_3, ..., x_n: real numbers)
begin
   for i := 1 to n − 1 do
      for j := n downto i + 1 do
         if x_j < x_{j−1} then
            begin        {interchange}
               temp := x_{j−1}
               x_{j−1} := x_j
               x_j := temp
            end
end
```

Figure 10.2

Bubble Sort (cont.)

- Example:

i = 1									
x_1	7		7		7		7	j = 2	2
x_2	9		9		9	j = 3	2		7
x_3	2		2	j = 4	2		9		9
x_4	5	j = 5	5		5		5		5
x_5	8		8		8		8		8

Four comparisons and two interchanges.

i = 2							
x_1	2		2		2		2
x_2	7		7		7	j = 3	5
x_3	9		9	j = 4	5		7
x_4	5	j = 5	5		9		9
x_5	8		8		8		8

Three comparisons and two interchanges.

i = 3					
x_1	2		2		2
x_2	5		5		5
x_3	7		7	j = 4	7
x_4	9	j = 5	8		8
x_5	8		9		9

Two comparisons and one interchange.

i = 4		
x_1	2	
x_2	5	
x_3	7	
x_4	8	j = 5
x_5	9	

One comparison but no interchanges.

Figure 10.3

Idea of Merge Sort

- Ex 12.16: Sort 6,2,7,3,4,9,5,1,8 by dividing them into equal size sublists, and merge them backwards

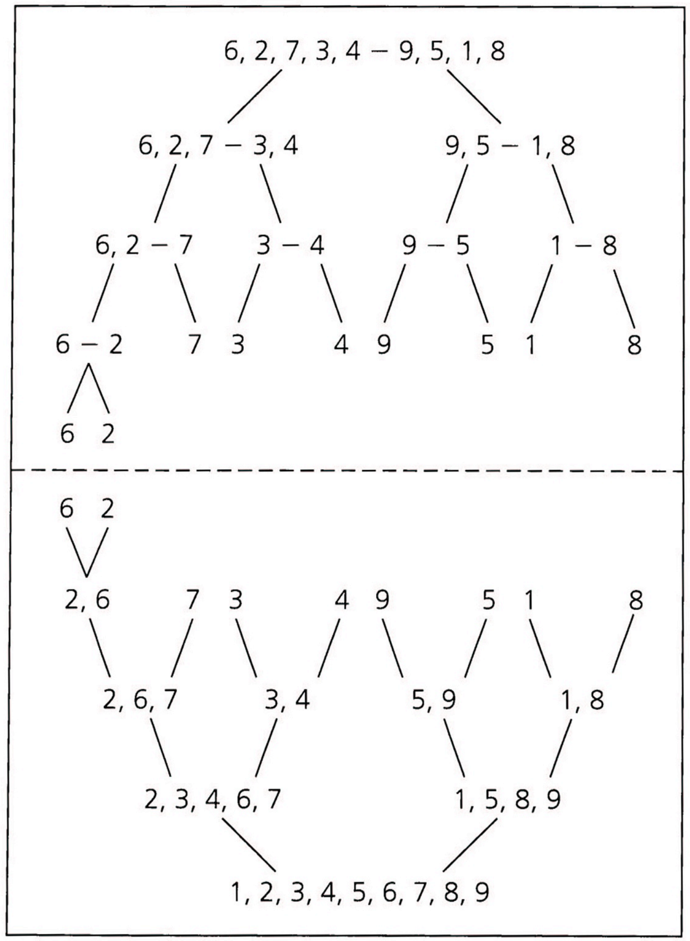


Figure 12.33

Each Merge Operation

- Before we quantify the complexity, first calculate the complexity of each merge
- Let L_1 and L_2 be the two sorted number, where L_1 has n_1 elements and L_2 has n_2. Merging L_1 and L_2 into another list consumes at most n_1+n_2-1 comparisons $\leftarrow O(n)$
- $L=Merge(L_1, L_2)$
  - 1: Let L be empty set
  - 2: Compare the first elements of L_1 and L_2, remove the smaller one and put it at the end of L
  - 3: If one of L_1 and L_2 is empty, append the other one to L. Otherwise go back to 2

Merge Sort

- 1: Divide the input array into two sublists L_1 and L_2, each has $\lfloor \frac{n}{2} \rfloor$ elements
- 2: Call MergeSort with L_1 and L_2
- 3. Merge(L_1, L_2)

- At most $\log_n$ levels, so the total complexity is $O(n \log_n)$

Outline

12.1 Definitions, Properties, and Examples

12.2 Rooted Trees

12.3 Trees and Sorting

12.4 Weighted Trees and Prefix Codes

12.5 Biconnected Components and Articulation Points

Codes

- Fixed-length versus variable-length codes
- Why do we need variable-length codes?
  - (English) letters appears in different frequencies → Assigning shorter code to more frequent letter results in shorter coded words
- For example, consider a set S={a,e,n,r,t} and code a:01, e:0, n:101, r:10, r:1, what is the coded word of "ata"?
  - Problem, this coded words also means "eta", "atet", and "an"
  - Why?

Unambiguous Codes

- Consider a different code a:01, e:0, n:101, r:10, r:1, what is the coded word of “ata”?
  - No ambiguity

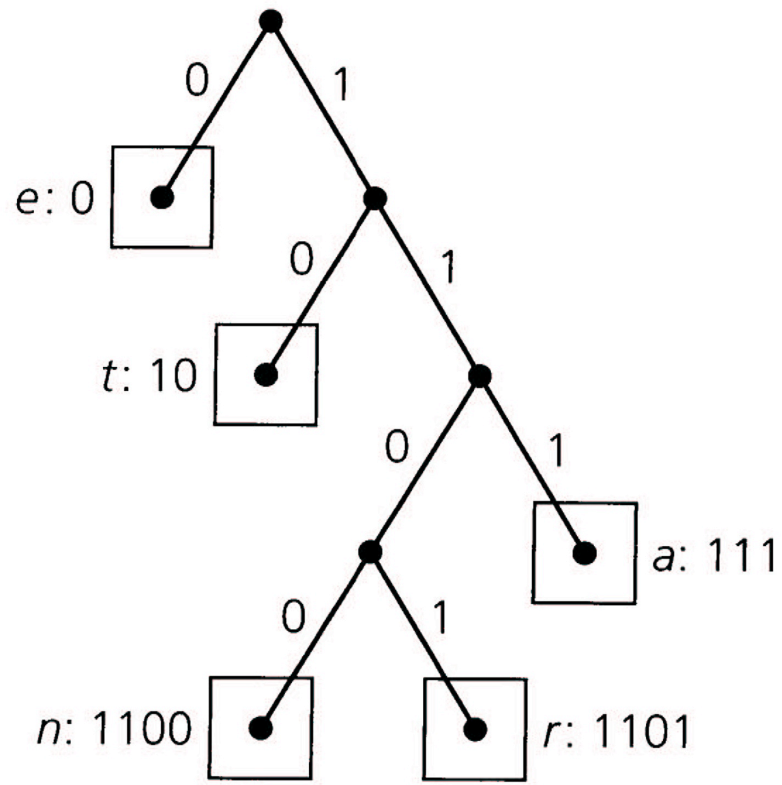


Figure 12.34

Prefix Code

- A set P of binary sequences is called a prefix code if no sequence in P is the prefix of any other sequence in P
- How to determine whether P is a prefix code?
- T is a full binary tree of height h if all the leaves are at level h

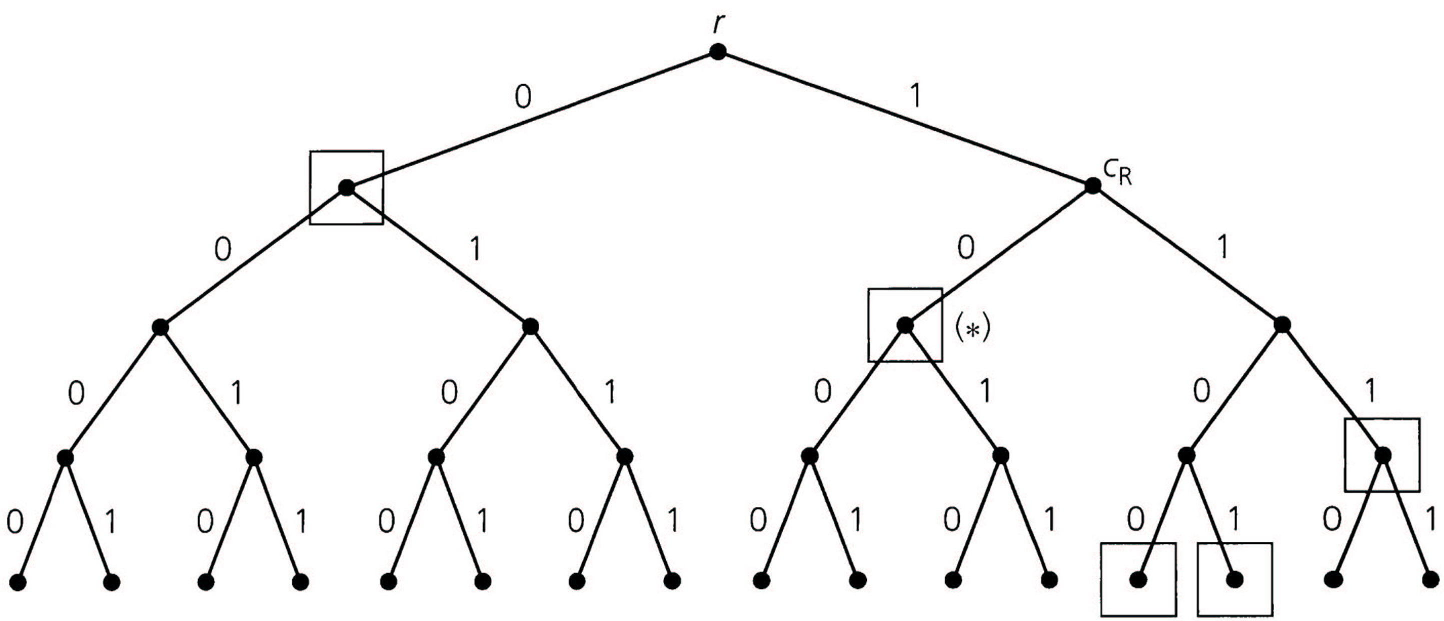


Figure 12.35

Efficient Code

- Lemma: If T is an optimal tree for $w_1 <= w_2 <= \ldots <= w_n$, there exists an optimal tree T', in which w_1 and w_2 are siblings at the maximal level of T'
  - Pushing w1 and w2 to the bottom couldn't be worse

- Theorem: Let T be an optimal tree with weight w_1+w_2, w_3, …, w_n, where $w_1 <= w_2 <= \ldots <= w_{n.}$ Dividing the leaf w_1+w_2 into two leavesw_1, w_2 results in a new optimal tree T'
  - Proved by the fact that there are only finite number of complete binary trees

Huffman Code

- A systematic way to create an efficient code
  - Create n active vertices each with a weight
  - Repeatedly find the two smallest active vertices with weights w_i and w_j, make them inactive, create a new active internal vertex to be their parent, and assign weight w_i+w_j.
  - Stop until there is only one active vertex
- Get the Huffman code by traversing from root to each leaf
- Ex 12.18: Construct a Huffman code for the symbols a,o,q,u,y,z with frequencies 20,28,4,17,12,7. Find a Humffman code for them.

Outline

12.1 Definitions, Properties, and Examples

12.2 Rooted Trees

12.3 Trees and Sorting

12.4 Weighted Trees and Prefix Codes

12.5 Biconnected Components and Articulation Points

Articulation Point

- A vertex v in a loop-free undirected graph G=(V,E) is called an articulation point if $\kappa(G - v) > \kappa(G)$; i.e., G-v has more components than G
- A graph with no articulation points is called biconnected
- A maximal biconnected subgraph is called a biconnected component
  - A subgraph that is not contained in a larger subgraph

Example

- Articulation points: c,f, and four biconnected components

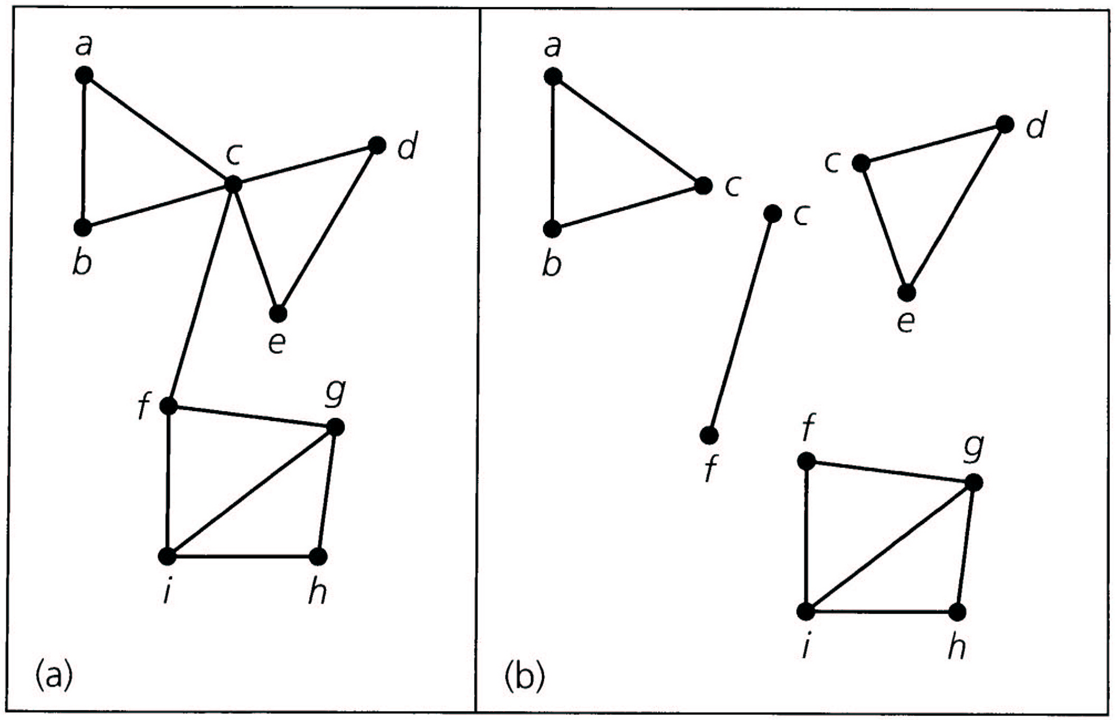


Figure 12.39

- How to systematically find the articulation points?

First Lemma

- A vertex z in G=(V,E) is an articulation point iff for any two vertices x,y where x, y, and z are not mutually equal, every path between x and y must go through z

Second Lemma

- Let G=(V,E) be a loop-free connected undirected graph, with a depth-first spanning tree T=(V,E'). If {a,b} is in E but is not in E', then a is either an ancestor or a descendant of b in tree T
- Proof Sketch: this is true otherwise {a,b} would be picked by the DFS algorithm (and thus is in E')
- Edges like {a,b} is called back-edge. So any edge in G is either: (i) an edge in T or (ii) an back-edge in it

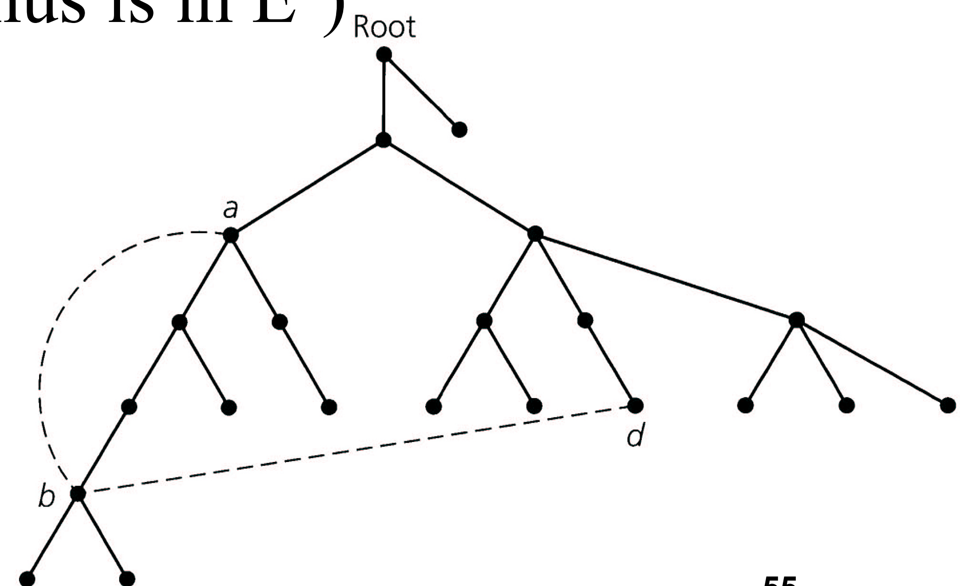


Figure 12.40

Third Lemma

- Let G=(V,E) be a loop-free connected undirected graph, with a depth-first spanning tree T=(V,E'). If r is the root of T, then r is an articulation point of G iff r has at least two children in T.
- Proof Sketch: If r has two children x_1 and x_2, and $\{x_1,x_2\}$ is not in E, then $\{x_1,x_2\}$ will be picked by the DFS algorithm

Fourth Lemma

- Let G=(V,E) be a loop-free connected undirected graph, with a depth-first spanning tree T=(V,E'). If v is a non-root vertex in T. v is an articulation point of G iff there exists a child c of v with no back-edge from a vertex z in the subtree rooted at c to a, which is an ancestor of v

Some Notations

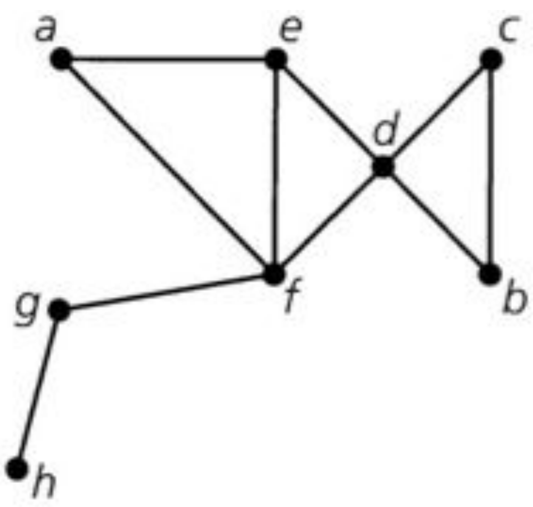
- We let dfi(x) be the depth-first index of x in preorder
  - If y is a descendant of x, then dfi(x)<dfi(y)
- We define low(x)=min{dfi(y)|y is adjacent to either x or a descendant of x in G} ← how to do this efficiently?
- If z is the parent of x (in T), compare low(x) and dfi(z)
  - low(x)=dfi(z): there is no vertex adjacent to an ancestor of z (via back-edge), so z is an articulation point
  - low(x)<dfi(z): there is a (some) descendant of z that is joined to an ancestor of z via a back-edge, so z is not an articulation point

Algorithm

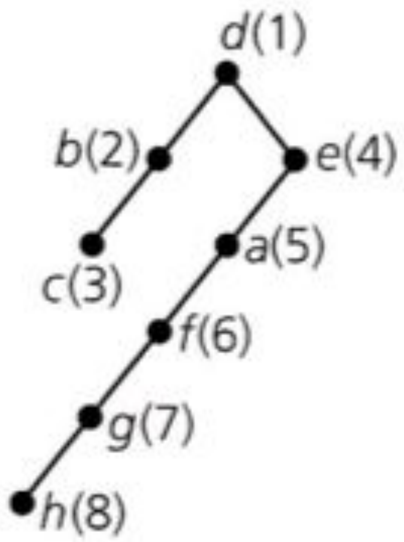
- 1: Let x_1, x_2, …, x_n be the vertices ordered by tree T
- 2: For $j=x_n, x_{n-1},\ldots,x_1$, compute $low(x_j)$ as follows
  - Let $low'(x_j)=\min\{dfi(z)|z$ is adjacent to x in G$\}$
  - Let $c_1,c_2,\ldots,c_m$ are the children of x_j, $low(x_j)=\min(low'(x_j),low(c_1),\ldots,low(c_m)\}$
- 3: For w_j, the parent of x_j, if $low(x_j)=dfi(w_j)$, then w_j is an articulation point of G unless w_j is the root and w_j has only one child (which is x_j).
  - The subtree rooted at x_j with $\{w_j,x_j\}$ is a biconnected component of G

A Complete Example

- Ex 12.20: Find the articulation points of G
- Step 1: First create a DFS tree, numbers in parentheses represent the dfi



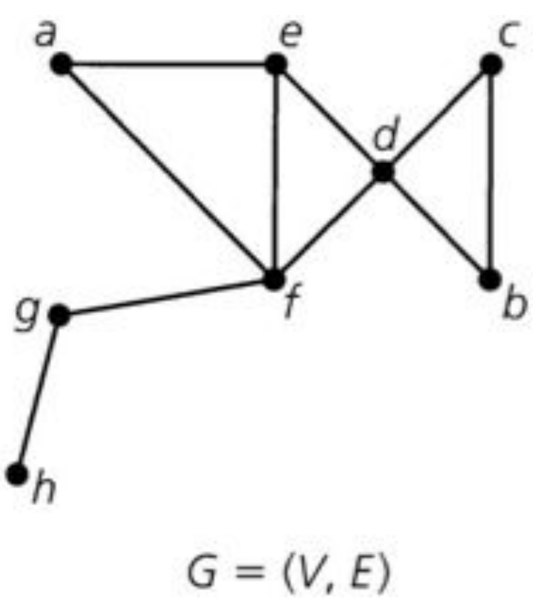
$G = (V, E)$

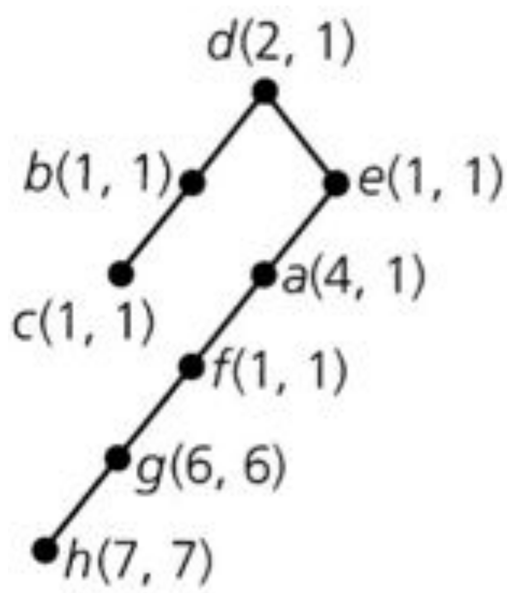


$T = (V, E')$

A Complete Example (cont.)

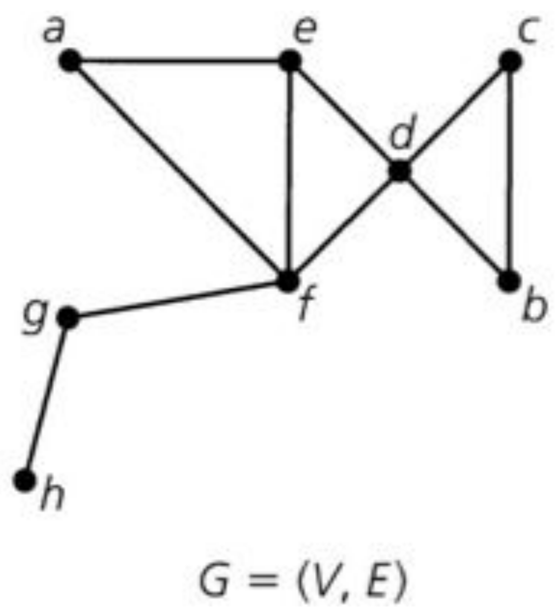
- Step 2: Compute (low’(x), low(x)), from bottom to up

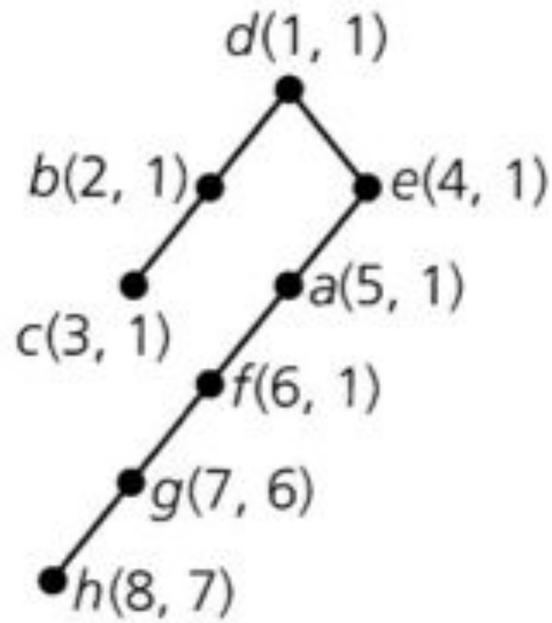




A Complete Example (cont.)

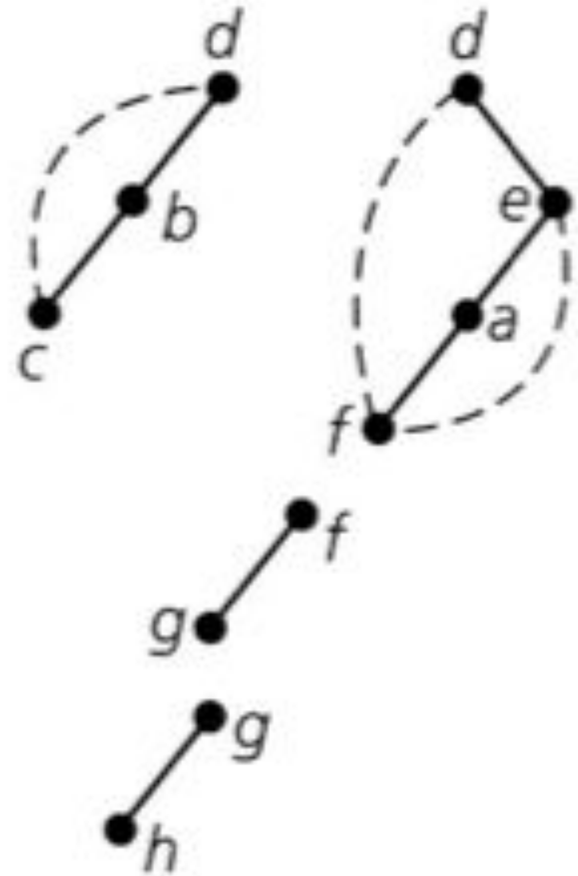
- Step 3: Compare (dfi(x), low(x))





A Complete Example (cont.)

- Last, we get the articulation points: g, f, d and four biconnected components



Take-home Exercises

- Exercise 12.1: 1, 2, 6, 13, 18
- Exercise 12.2: 1, 3, 5, 9, 12, 17
- Exercise 12.3: 1, 2, 3
- Exercise 12.4: 1, 3, 5, 7
- Exercise 12.5: 1, 2, 10