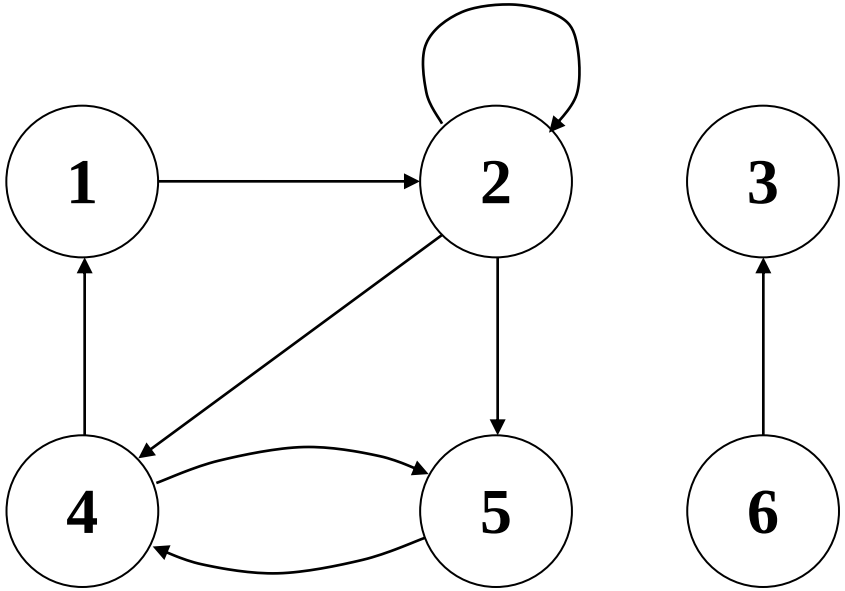
Definition of Graphs and Trees. Representation of Trees.

Chapter 6

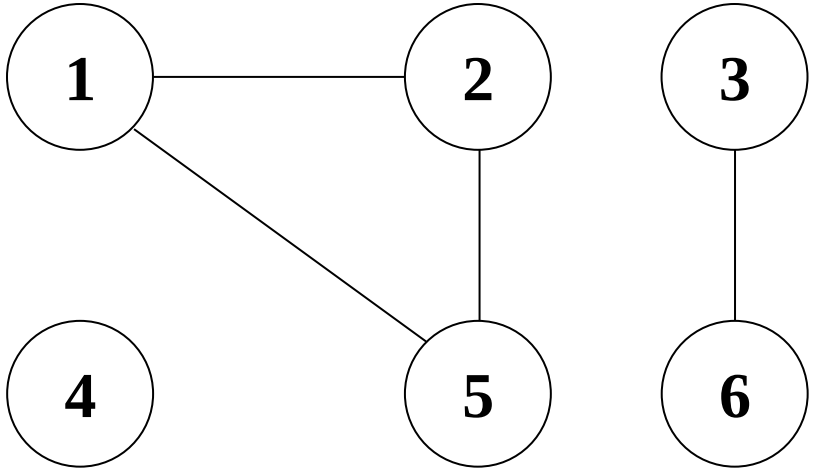
2016

Definition of graphs (I)

- A *directed graph* or *digraph* is a pair $G = (V,E)$ s.t.:
  - V is a finite set called the *set of vertices* of G.
  - $E \subseteq V \times V$ is a binary relation on V called the *set of arcs* of G. An *arc* of G is denoted by an ordered pair of vertices (u,v), $u,v \in V$. Note that $(u,v) \neq (v,u)$.
- An *undirected graph* is a pair $G = (V,E)$ s.t.:
  - V is a finite set called the *set of vertices* of G.
  - E is a set of unordered pairs of vertices $\{u,v\}$, $u,v \in V$ called the *edges* of G. For uniformity we denote an edge with (u,v), but in an undirected graph $(u,v) = (v,u)$.
- Note that we allow self-loops only in directed graphs.



a. A directed graph G_1

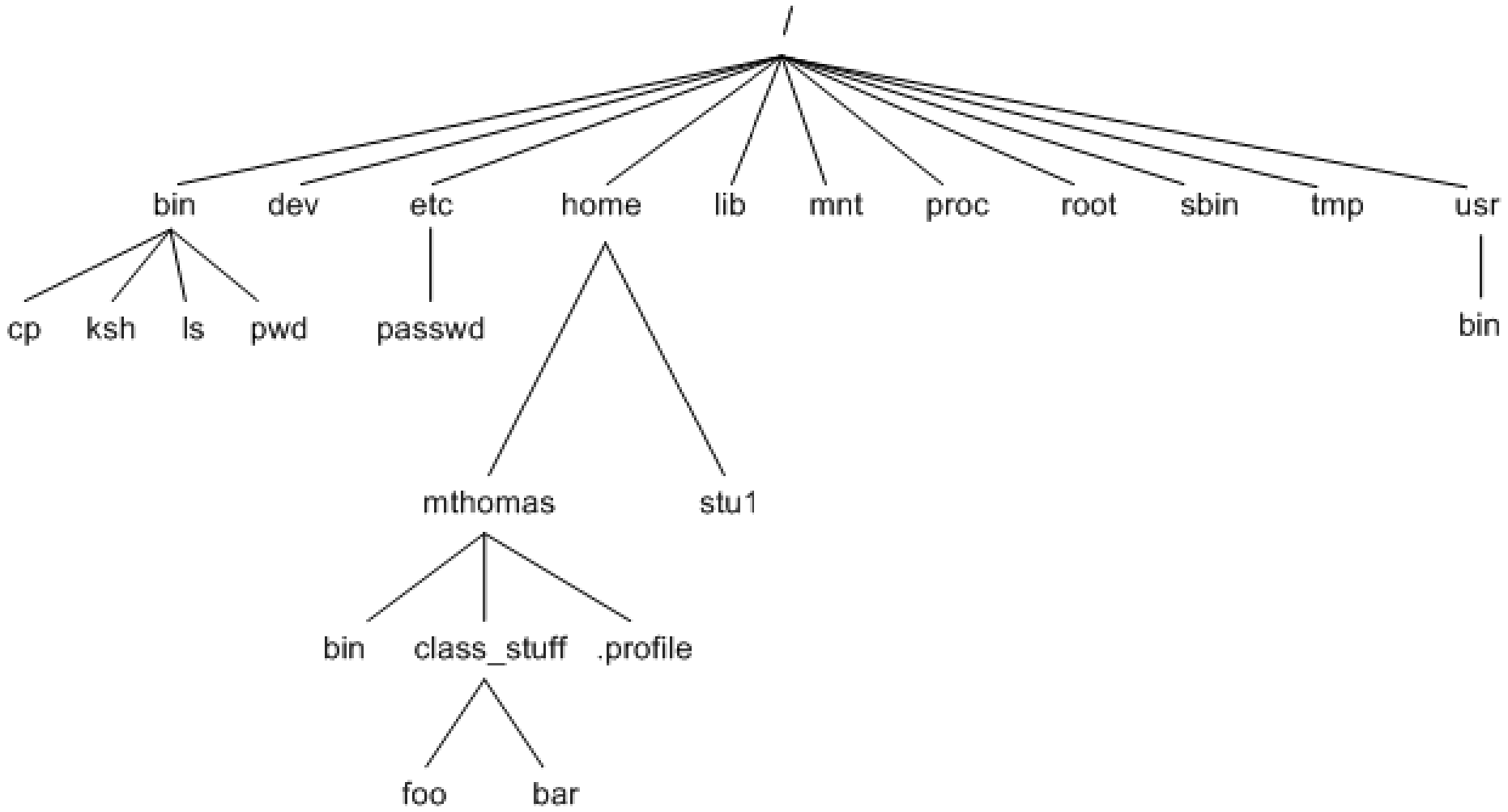


b. An undirected graph G_2

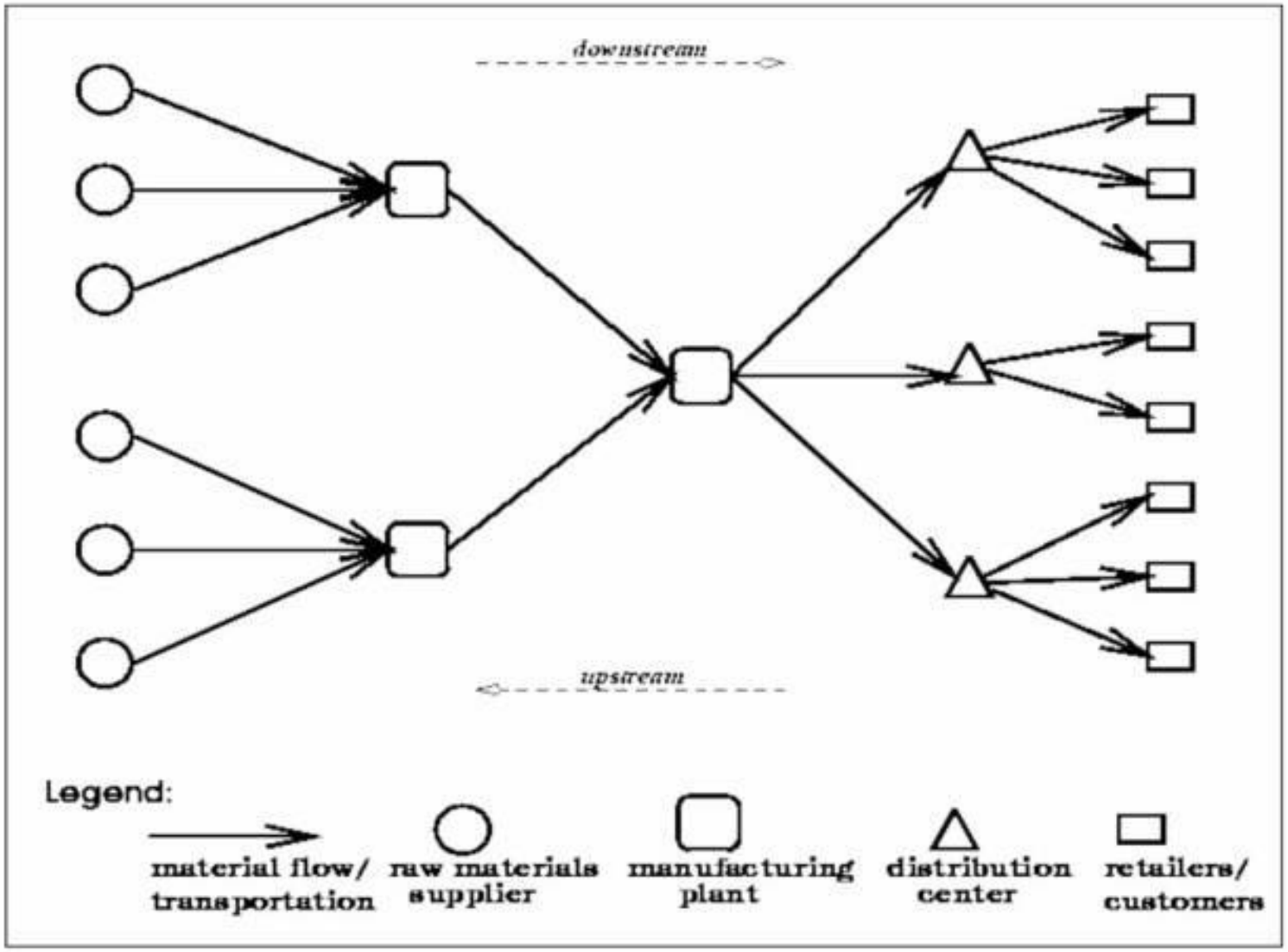
Definition of graphs (II)

- If $a = (u,v)$ is an arc in a directed graph then a is called *incident-from u* and *incident-to v*.
- If $e = (u,v)$ is an edge in an undirected graph then e is called *incident* to u and v.
- If (u,v) is an arc (or edge) in (directed or undirected) graph then v is called *adjacent* to u. Note that the adjacency relation is symmetric for undirected graphs. If the graph is directed then v is called *adjacent-from u* and u is called *adjacent-to v*.
- If $G = (V,E)$ is an undirected graph and $v \in V$ then $degree(v) = |\{e \in E \mid e$ is incident to $v\}|$. For example in G_2 we have $degree(5) = 2$.
- If $G = (V,E)$ is a directed graph and $v \in V$ then $in\text{-}degree(v) = |\{e \in E \mid e$ is incident-to $v\}|$ and $out\text{-}degree(v) = |\{e \in E \mid e$ is incident-from $v\}|$. For example in G_1 we have $in\text{-}degree(4) = 2$ and $out\text{-}degree(2) = 3$.
- If $G = (V,E)$ is a directed graph and $v \in V$ then $degree(v) = in\text{-}degree(v) + out\text{-}degree(v)$. For example in G_1 we have $degree(2) = 5$.

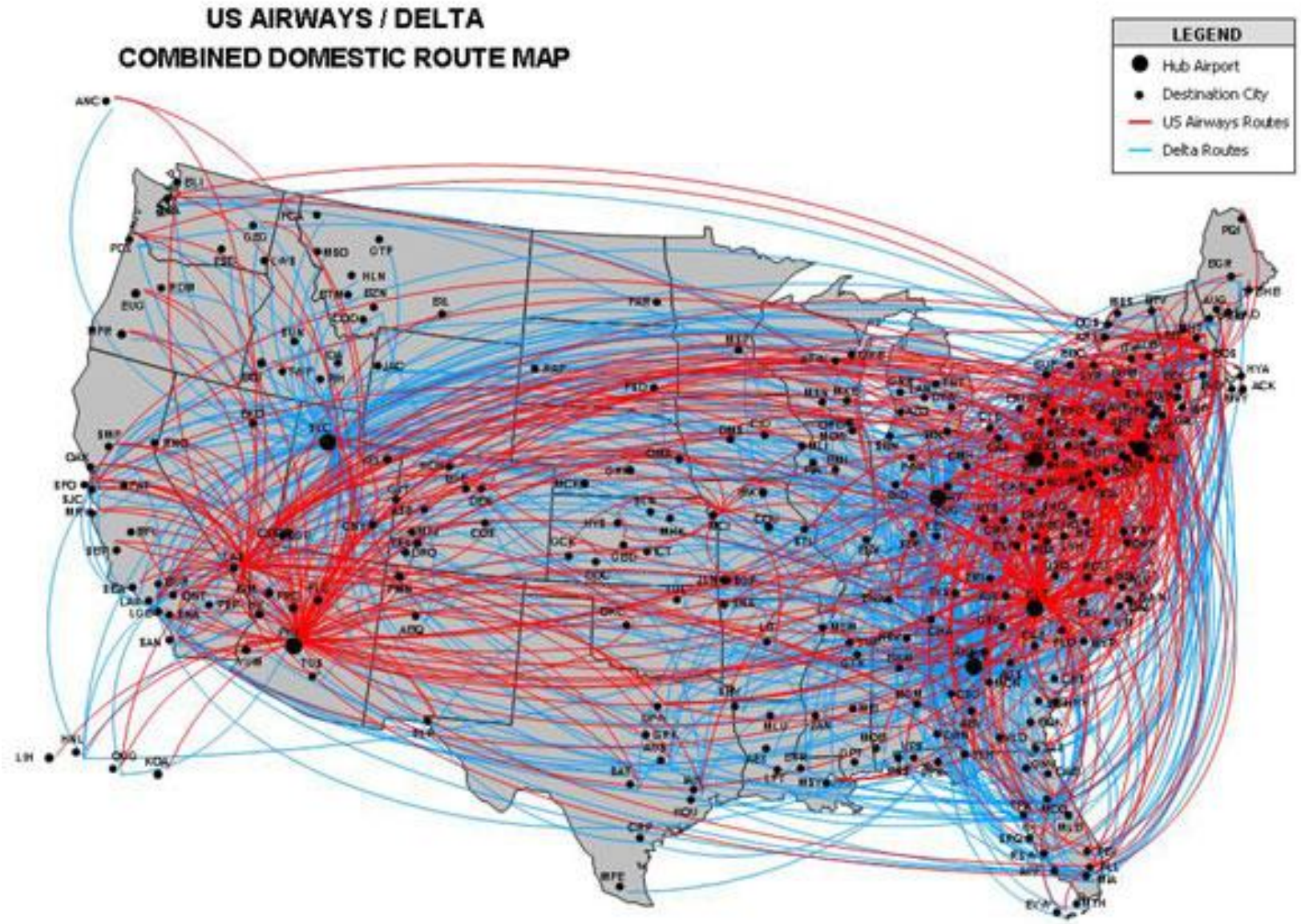
Unix file system graph



Supply chain graph

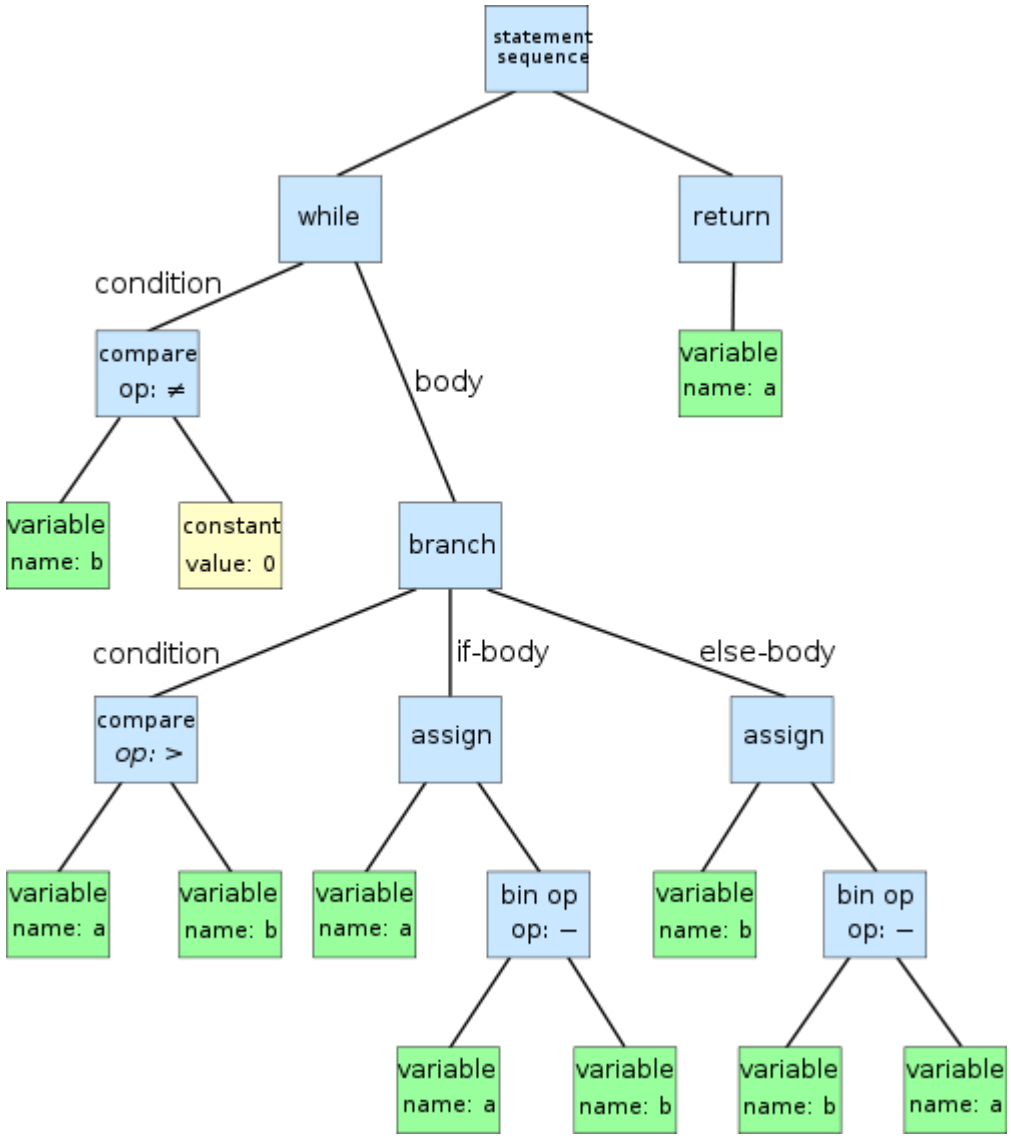


Airline networks

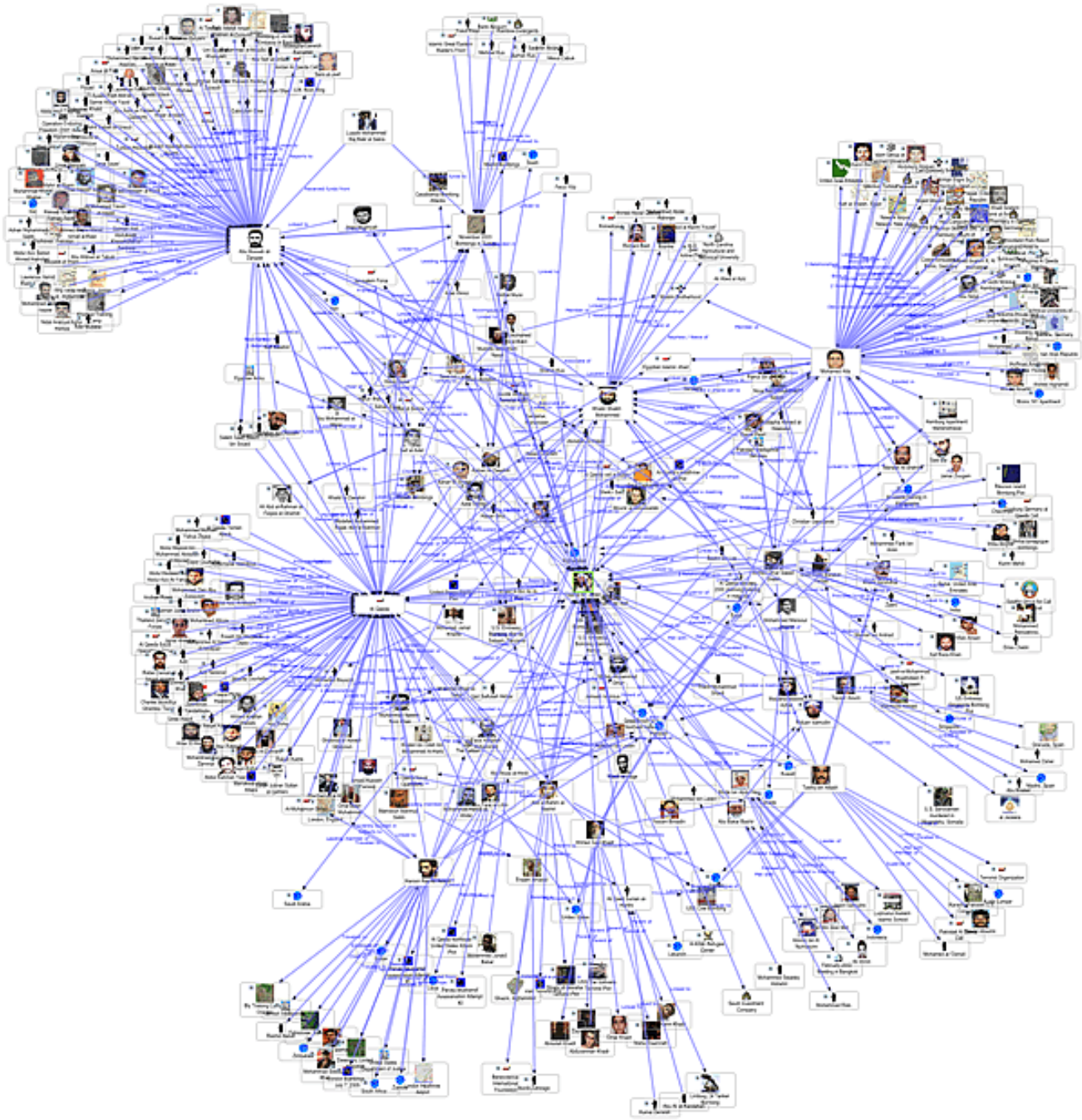


2016

Syntax tree



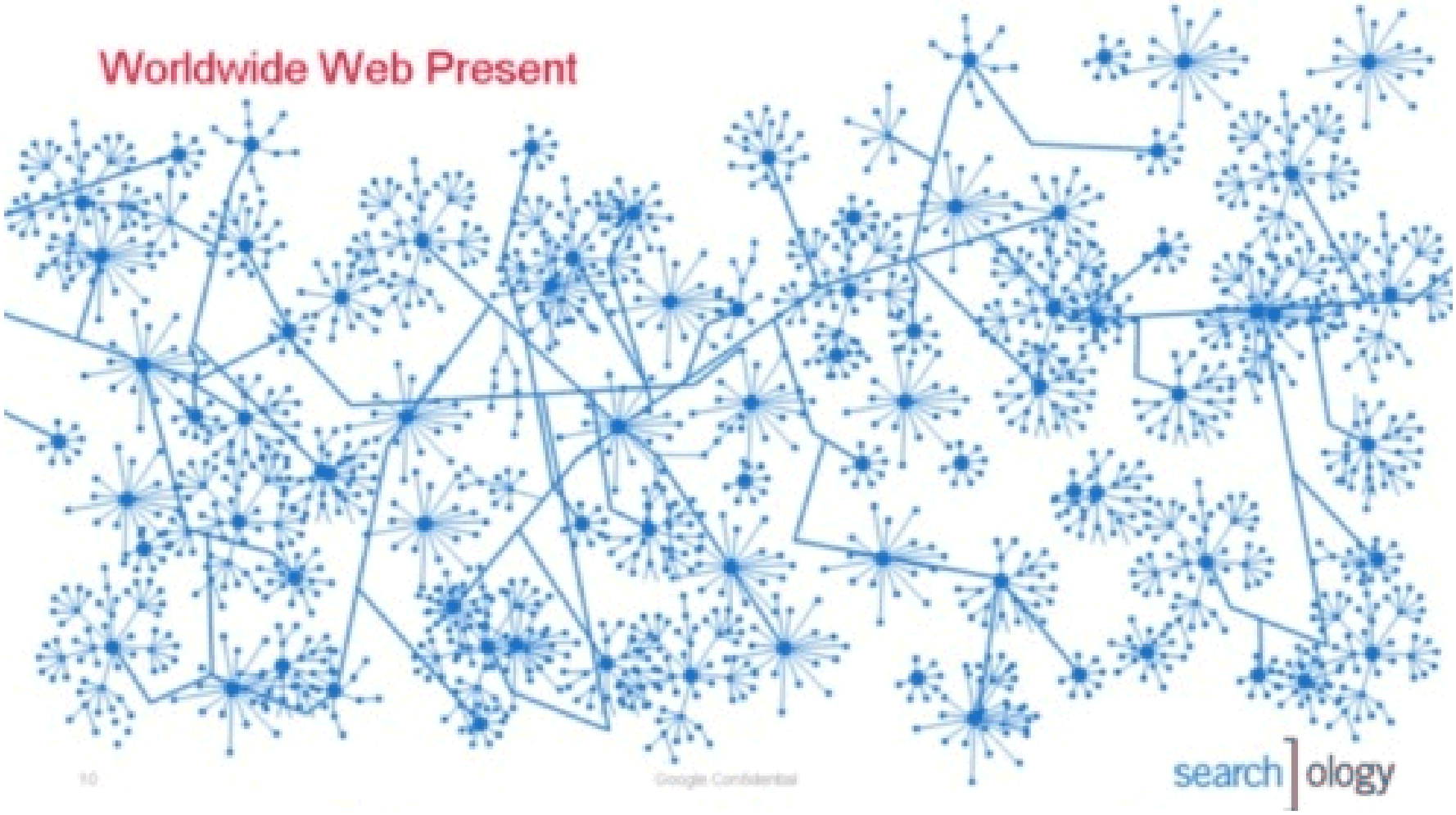

```
while b ≠ 0
      if a > b
            a := a − b
      else
            b := b − a
return a
```



Social network graph

2016

Web graph



2016

Paths in graphs (I)

- A *path of length k* from a vertex u to a vertex u' in a graph $G = (V,E)$ is a sequence of vertices $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ such that $u = v_0$, $u' = v_k$ and $(v_{i-1}, v_i) \in E$ for all $i = 1, 2, \ldots, k$. Note then the length of a path is equal to the number of edges (arcs) of the path.
- If there is path p from u to u' then we say that u' is *accessible* from u via p and we write this as $u \rightarrow^p u'$.
- A path is called an *elementary path* if and only if all the vertices on the path are distinct. For example, in G_1, $\langle 1, 2, 5, 4 \rangle$ is an elementary path and $\langle 2, 5, 4, 5 \rangle$ is not an elementary path.
- A *sub-path* of a path $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a contiguous subsequence of vertices in p. Thus, for all i and j s.t. $0 \leq i \leq j \leq k$ the sequence of vertices $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ is a sub-path of p.
- A *cycle* in a directed graph is a path $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ s.t. $v_0 = v_k$. A cycle is called an *elementary cycle* if and only if $v_1, v_2, \ldots v_k$ are all distinct. Note that a self-loop in a directed graph is a cycle of length 1.

Paths in graphs (II)

- Two paths $\langle v_0, v_1, v_2, \ldots, v_{k-1}, v_0 \rangle$ and $\langle w_0, w_1, w_2, \ldots, w_{k-1}, w_0 \rangle$ *are the same cycle* if and only if exists an integer j such that $w_i = v_{(i+j) \bmod k}$ for all $i = 0, 1, \ldots, k-1$. For example, in G_1 the path $\langle 2, 4, 1, 2 \rangle$ is the same cycle as $\langle 4, 1, 2, 4 \rangle$.
- A directed graph without self-loop is called an *elementary directed graph*.
- An *elementary cycle* in an undirected graph is a path $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ s.t. $v_0 = v_k$, $k \geq 3$ and $v_1, v_2, \ldots v_k$ are all distinct. For example, in G_2 the path $\langle 1, 2, 5, 1 \rangle$ is an elementary cycle and $\langle 1, 5, 1 \rangle$ is a cycle which is not elementary.
- A graph without cycles is called *acyclic*. A *directed acyclic graph* is sometimes called a DAG.

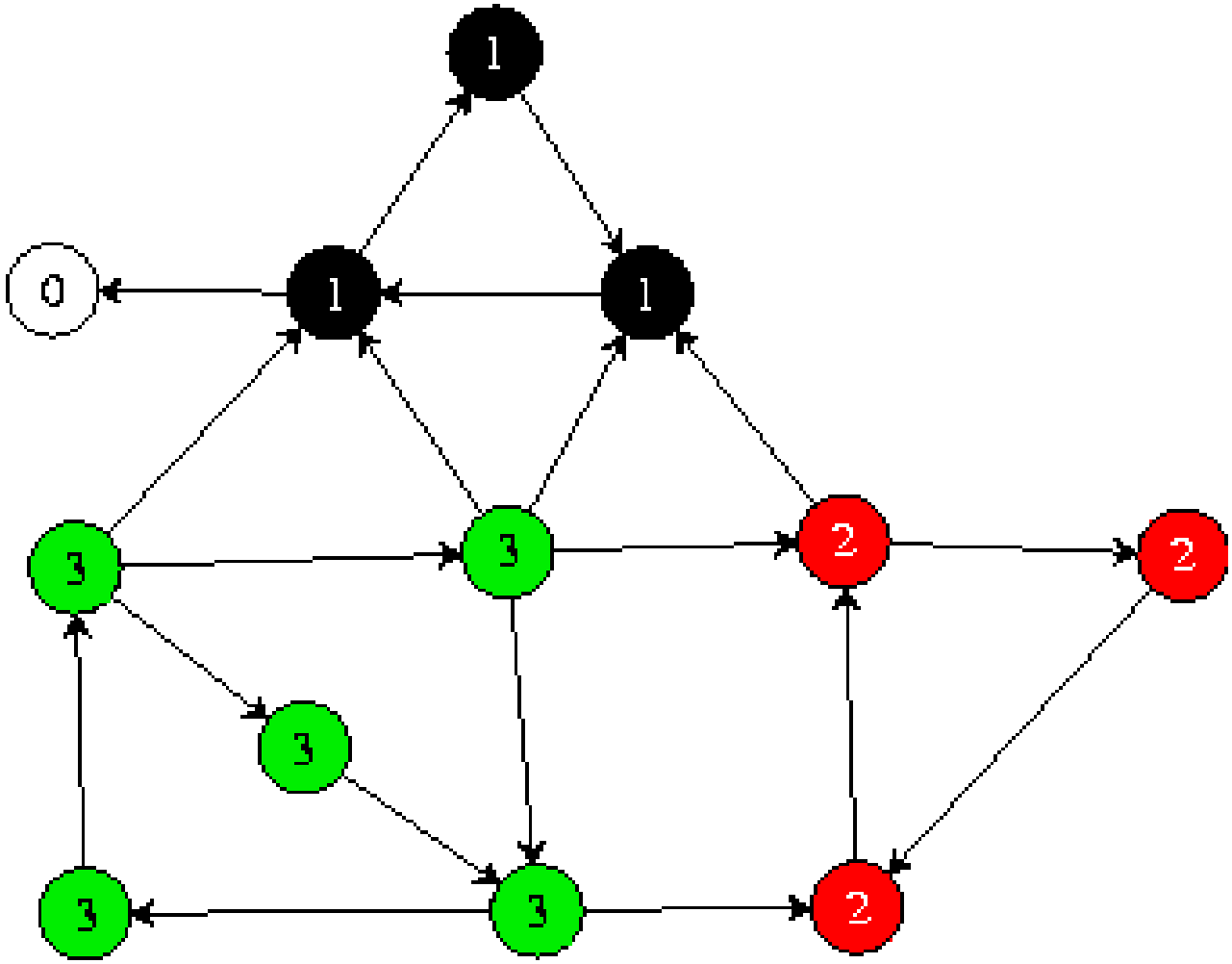
Connected graphs

- An undirected graph is called *connected* if and only if for all pairs of vertices there is a path that connects them.
- Property 1: the accessibility relation between vertices of an undirected graph is an *equivalence relation*, i.e. it is *reflexive*, *symmetric* and *transitive*. Prove this statement as homework.
- The *connected components* of an undirected graph are the equivalence classes defined by the accessibility relation. For example, the connected components of G_2 are {1, 2, 5}, {3, 6} and {4}. An undirected graph is connected if and only if it has a single connected component.

Strongly connected graphs

- A directed graph is called *strongly connected* if and only if for all pairs of vertices, each one is accessible from the other.
- Property 2: the mutual accessibility relation between the vertices of a directed graph is an equivalence relation. Prove this statement as homework.
- The *strongly connected components* of a directed graph are the equivalence classes defined by the mutual accessibility relation. For example, the strongly connected components of G_1 are: {1, 2, 4, 5}, {3} and {6}. A directed graph is strongly connected if and only if it has a single strongly connected component.

Strongly connected components - example



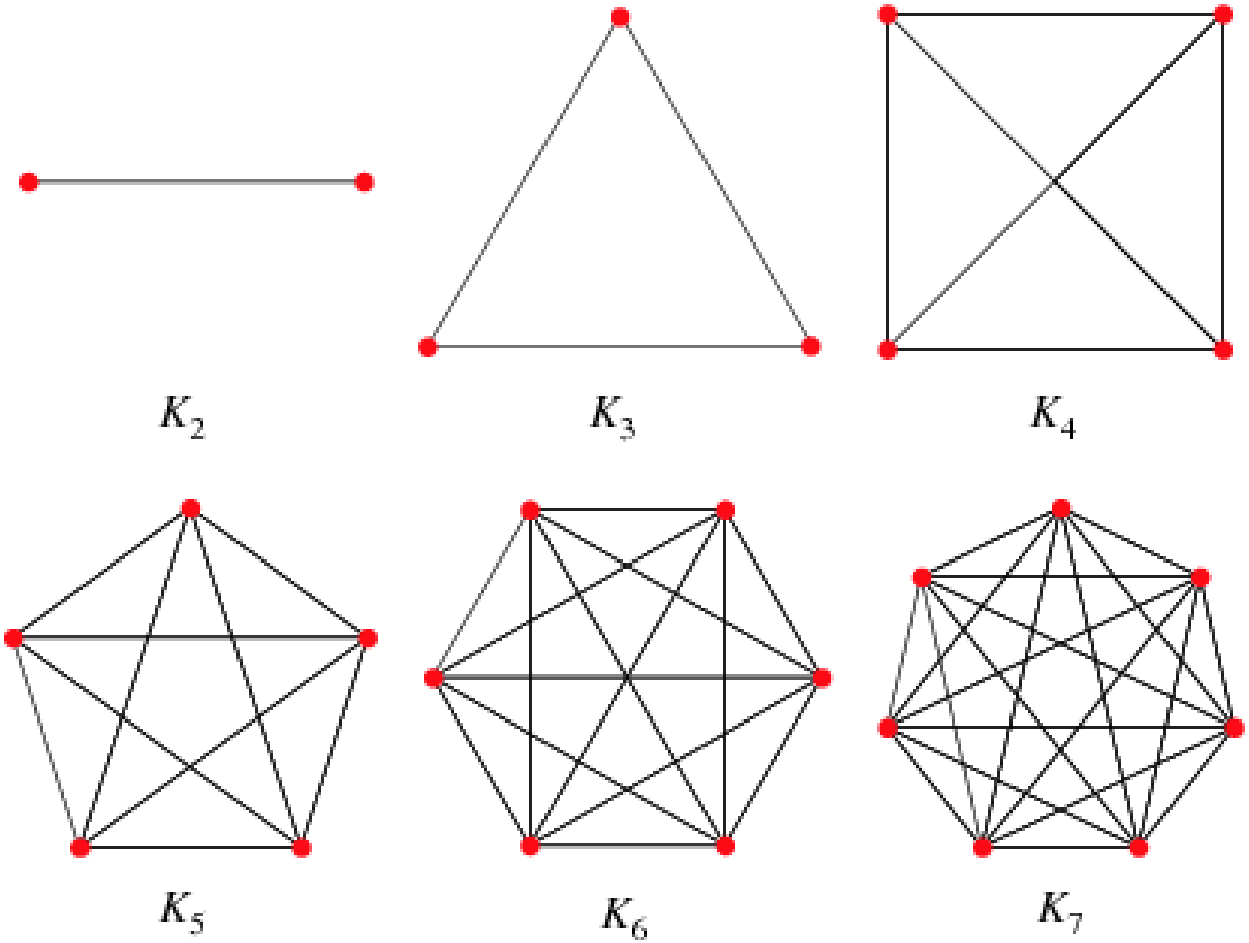
Sub-graphs

- Graph $G' = (V',E')$ is *sub-graph* of graph $G = (V,E)$ if $V' \subseteq V$ and $E' \subseteq E$.
- If $V' \subseteq V$ then the sub-graph of G *induced by* V' is $G' = (V',E')$ s.t.:

  $E' = \{(u,v) \in E \mid u,v \in V'\}$
- If $G = (V,E)$ is an undirected graph then the *directed version* of G is $G' = (V',E')$ s.t. $(u,v) \in E'$ if and only if $(u,v) \in E$. This means that each edge (u,v) in G is substituted by two arcs (u,v) and (v,u) in G'.
- If $G = (V,E)$ is a directed graph then the *undirected version* of G is $G' = (V',E')$ s.t. $(u,v) \in E'$ if and only if $u \neq v$ and $(u,v) \in E$. This means that the undirected version is obtained from the directed version by eliminating directions and self-loops. Note that because (u,v) and (v,u) represent the same edge of an undirected graph, the undirected version of a directed graph contains it only once.
- In a directed graph, the *neighbor* of a vertex u is any vertex adjacent to u in the undirected version of the graph.

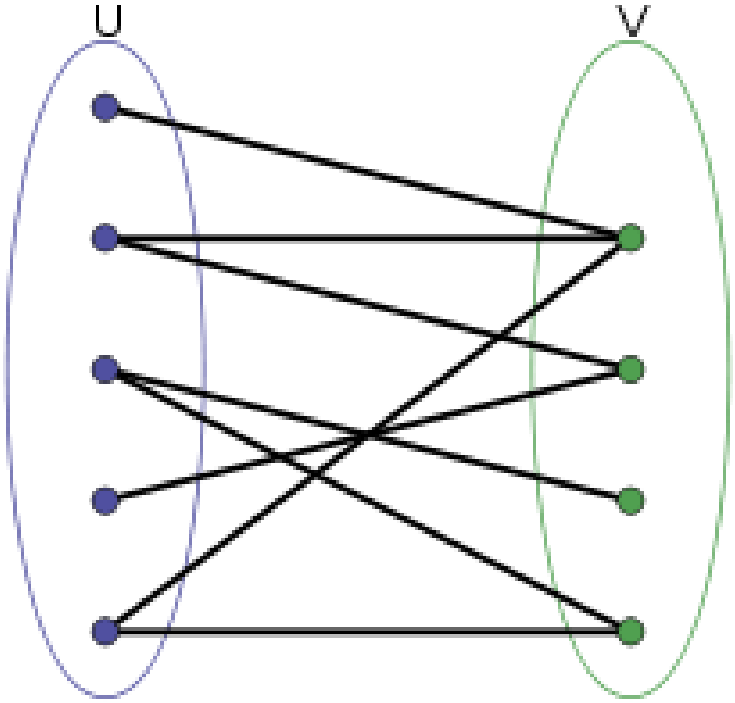
Complete graphs

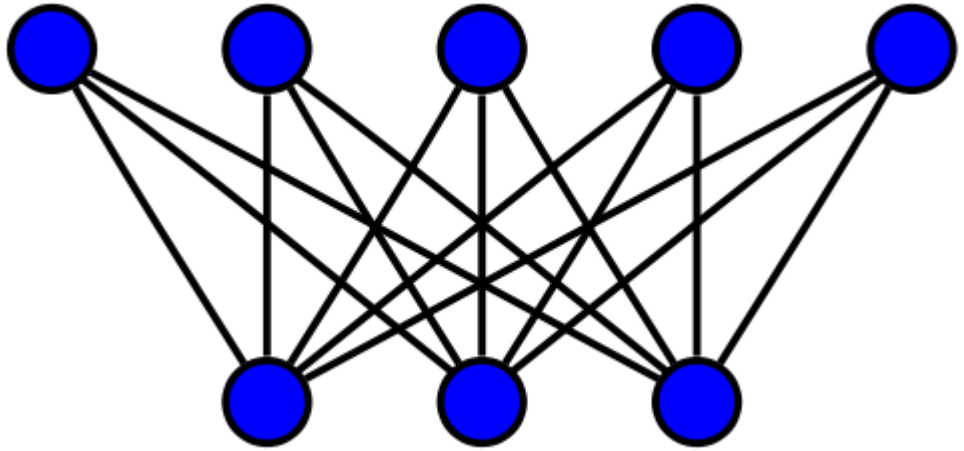
- A *complete graph* (or *clique*) is an undirected graph $G = (V,E)$ s.t. $E = \{(u,v) \mid u \neq v \text{ and } u,v \in V\}$.



Bipartite graphs

- A *bipartite graph* is an undirected graph G s.t. *its set of vertices* can be partitioned into two sets U and V s.t. if $(u,v) \in E$ then either $u \in U$ and $v \in V$ or $u \in V$ and $v \in U$.

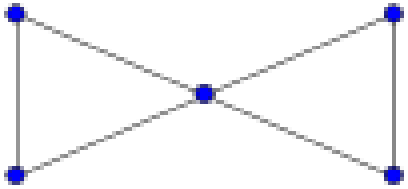




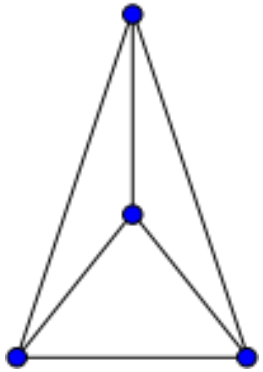
Complete bipartite graph (*biclique*) $K_{5,3}$

Planar graphs

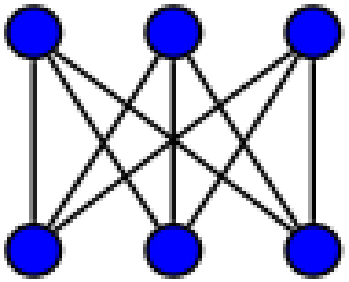
- A *planar graph* is a graph that *can be embedded in the plane*, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints.
- Kuratowski's theorem: A finite graph is planar if and only if it does not contain a subgraph that is a subdivision of K_5 or $K_{3,3}$.
- A *subdivision* of a graph results from inserting vertices into edges (for example, changing an edge •——• to •—•—•) zero or more times.
- Practical criteria (theorems):
  - If $v \geq 3$ then $e \leq 3v - 6$;
  - If $v \geq 3$ and there are no cycles of length 3, then $e \leq 2v - 4$.



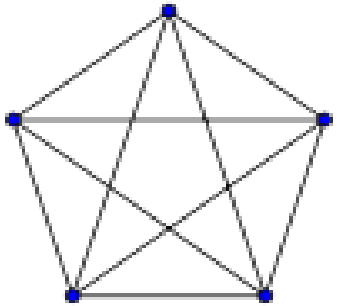
Butterfly graph



K_4



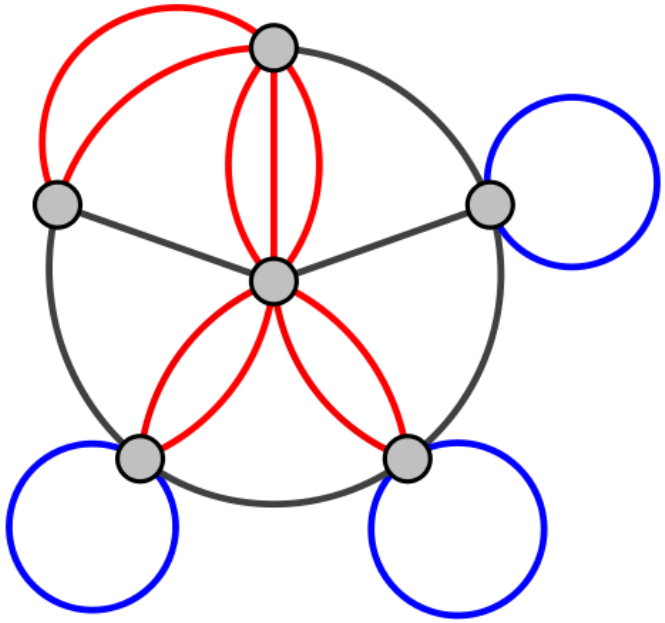
$K_{3,3}$

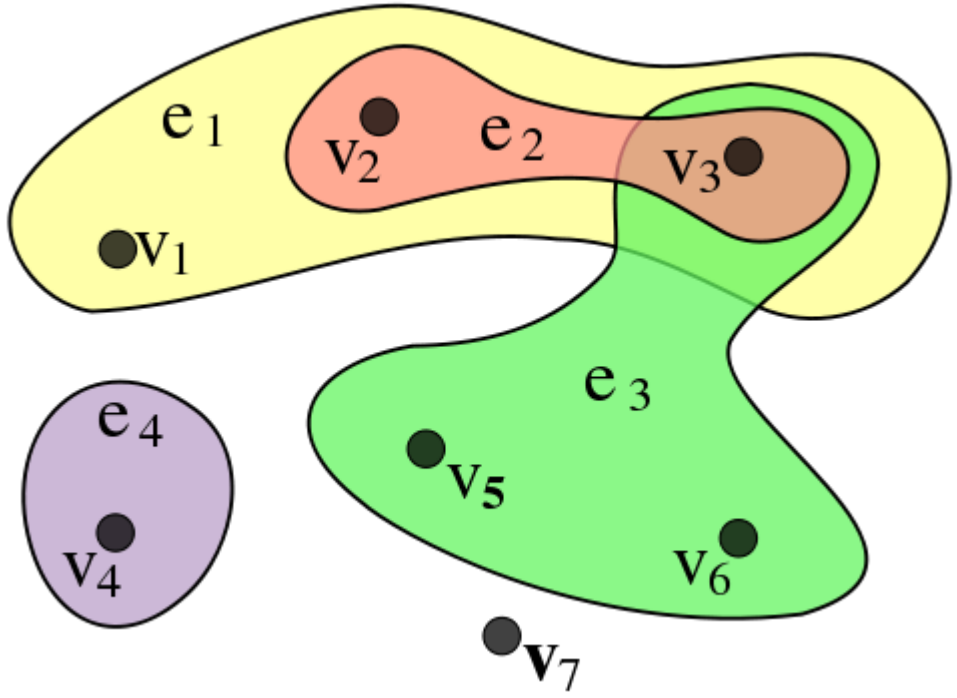


K_5

Multi-graphs and hyper-graphs

- A *multi-graph* is derived from an undirected graph by allowing self-loops and multiple edges between its vertices.
- A *hyper-graph* is derived from an undirected graph by allowing an edge to connect an arbitrary subset of vertices rather than only two vertices. The edges of a hyper-graph are called *hyper-edges*.





Isomorphic graphs

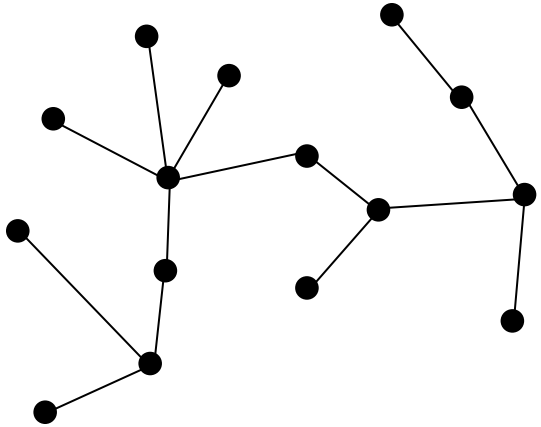
- Graphs $G = (V,E)$ and $G' = (V',E')$ are called *isomorphic* if and only if there is a one-to-one mapping $f : V \rightarrow V'$ s.t. $(u,v) \in E$ if and only if $(f(u),f(v)) \in E'$, i.e. G' can be obtained from G by renaming its vertices.

Forests

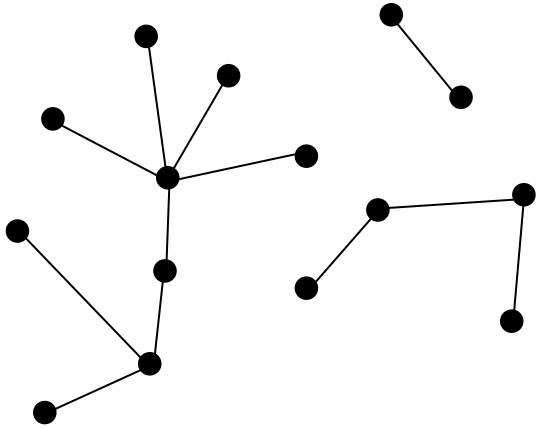
- An undirected acyclic graph is called a *forest*.
- A connected forest is called a (*free*) *tree*.
- A forest is composed of trees.

Free trees

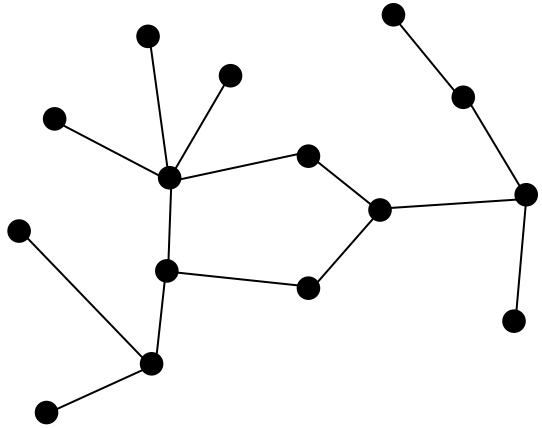
- There are many variations of the concept of tree: free trees, rooted trees, ordered trees and positional trees.
- A *free tree* is an undirected, acyclic and connected graph. If the connectedness property is dropped out then the graph is called a forest.



A free tree



A forest



An undirected graph which is neither a free tree nor a forest

Properties of free trees

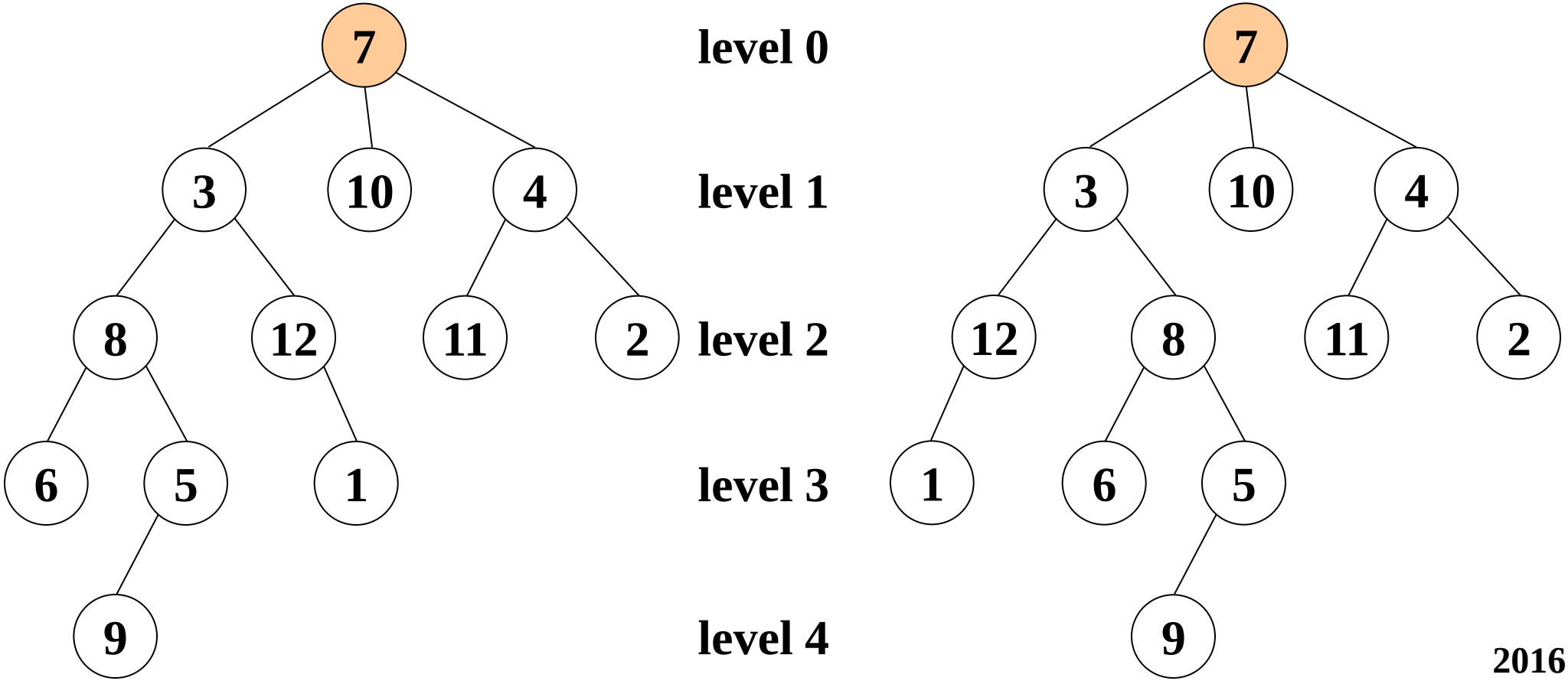
- Theorem: Let $G = (V,E)$ be an undirected graph. The following statements are equivalent:
  1. G is a free tree.
  2. Any two vertices of G are connected by a unique elementary path.
  3. G is connected but if we remove an arbitrary edge from E the resulting graph is not connected.
  4. G is connected and $|E| = |V| - 1$.
  5. G is acyclic and $|E| = |V| - 1$.
  6. G is acyclic but if we add an arbitrary edge to E the resulting graph contains a cycle.
- Proof: see the textbook. It follows the pattern: $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4 \Rightarrow 5 \Rightarrow 6 \Rightarrow 1$.
- Example proof for $6 \Rightarrow 1$: Let u and v be two arbitrary vertices of G. If they are adjacent, there is a path from u to v. If they are not adjacent then adding the edge (u,v) to E according to the hypothesis, the resulting graph will contain a cycle. The edges of this cycle that are distinct from (u,v) are all members of E and determine a path from u to v. Because u and v have been chosen arbitrarily, it follows that G is connected. But because according to the hypothesis G is acyclic, it follows that G is a free tree, q.e.d.

Rooted trees (I)

- A *rooted tree* is a free tree with a distinguished vertex called *root*. Vertices of trees are very often called *nodes*. We shall use this terminology hereafter.
- Let T be a tree rooted at r and let x be a node in T. Any node y on the unique path from r to x is called an *ancestor* of x.
- If y is an ancestor of x then x is a *descendant* of y. Note that any node is both a descendant and an ancestor of itself.
- If y is an ancestor of x and $y \neq x$ then y is called a *proper ancestor* of x and x is called a *proper descendant* of y.
- The *sub-tree of T rooted at x* is the tree induced by the descendants of x and with root x.
- If the last edge on the path from r to x in T is (y,x) then y is called the *parent* of x and x is called the *child* of y. Note that the root r is the unique node of T with no parent.
- Two nodes with the same parent are called *siblings*. A node with no child is called an *external node* or *leaf*. Otherwise it is called *internal node*.

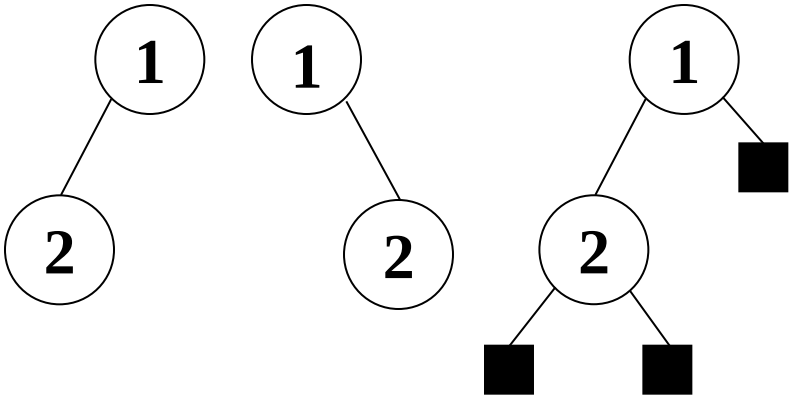
Rooted trees (II)

- The number of children of a node *x* of a rooted tree *T* is called the *degree* of *x*.
- The length of the unique path from the root *r* to a node *x* of a tree T is called the *depth* or *level* of *x* in *T*. The highest depth of a node of a tree is called the *height* of the tree.
- An *ordered tree* is a rooted tree s.t. for each node the set of its children is ordered. This means that if a node has *k* children then there is a first child, a second child, etc.



Binary trees

- The best way to define binary trees is using a *recursive definition*.
- A *binary tree* is :
  - Either an empty set of nodes defining an *empty binary tree*
  - Or a structure composed of a *root node*, a binary tree called *its left sub-tree* and a binary tree called *its right sub-tree*.
- In a non-empty binary tree with root *r*:
  - If the left sub-tree is non-empty then its root is called the *left child* of *r*
  - If the right sub-tree is non-empty then its root is called the *right child* of *r*
- Important note: a binary tree is NOT an ordered tree with the degree of its nodes less or equal than 2 because in an ordered tree if a node *x* has a single child then it cannot be qualified as left or right child of *x*, while in a binary tree this distinction is important ! Another distinction ?

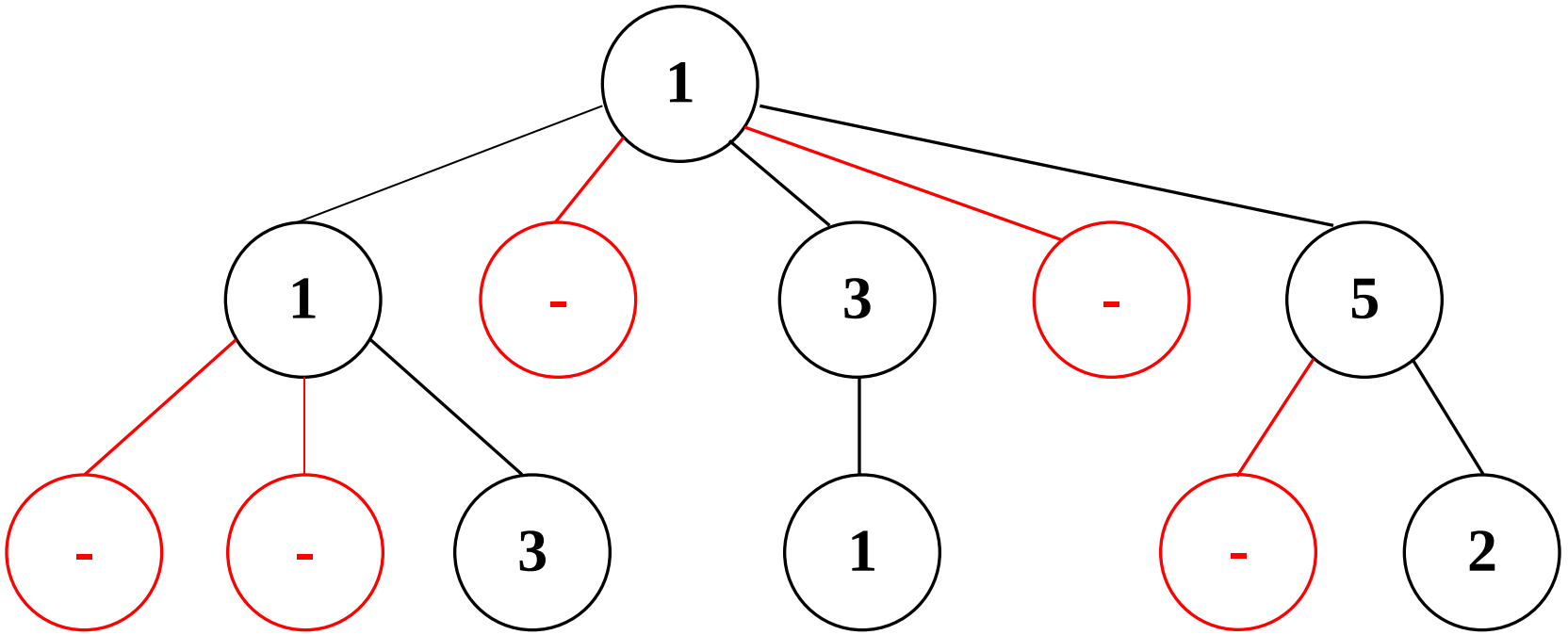


• These trees are identical as ordered trees but not as binary trees.

• A binary tree can be mapped to an ordered tree by adding an explicit representation of the missing information.

Positional trees

- In a *positional tree* the children of a node are labeled with distinct consecutive positive integer numbers. The *i*-th child of a node *x* is *absent* if there are no children of *x* labeled with *i*.



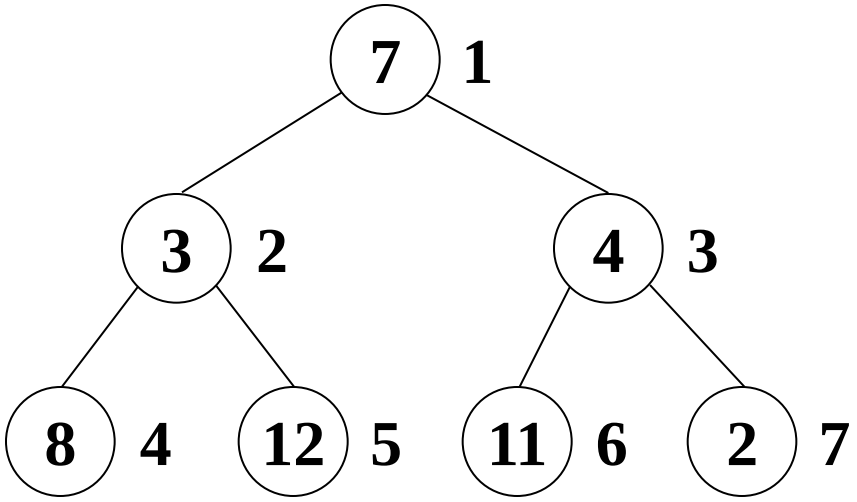
K-ary positional trees

- A *k-ary tree* is a positional tree such that for each node *x* the children labeled with values greater than *k* are missing. A *k*-ary tree with $k = 2$ is called a *binary tree*.
- A *strict k-ary tree* is a *k*-ary tree s.t. all the internal nodes have degree *k*. Note that adding the missing information we obtain a strict *k*-ary tree.
- A *full k-ary tree* is a strict *k*-ary tree s.t. all the leafs have the same depth.
- The number of nodes of depth *d* in a full *k*-ary tree is k^d. It follows that the height of a full *k*-ary tree with *n* leafs is $\log_k n$.
- The number of internal nodes of a full *k*-ary tree of height *h* is:

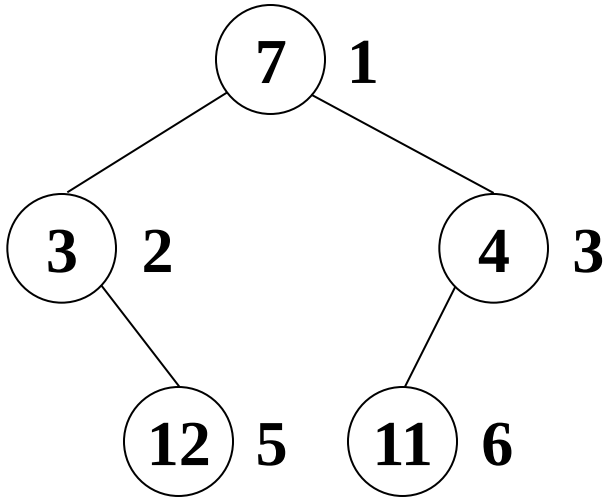
  $1 + k + k^2 + \ldots + k^{h-1} = (k^h - 1)/(k - 1)$
- The number of internal nodes of a full binary tree of height *h* is $2^h - 1$.

Representation of binary trees (I)

- A binary tree of height h can be represented using an array of size $2^{h+1}-1$ because it has maximum 2^h-1 internal nodes and 2^h leafs. This representation is straight forward, but has the drawback that if the tree is sparse it still consumes memory which is exponential in the height of the tree.
- The nodes of a full binary tree can be numbered as follows:
  - The root is numbered with 1
  - The left child of node i is numbered with $2i$ and the right child with $2i+1$
- The node numbered with i is stored on the position i in the array
- The numbering scheme allows the computation of the parent, left child and right child in $O(1)$ time. The parent of node i is $\lfloor i/2 \rfloor$.



7	1
3	2
4	3
8	4
12	5
11	6
2	7



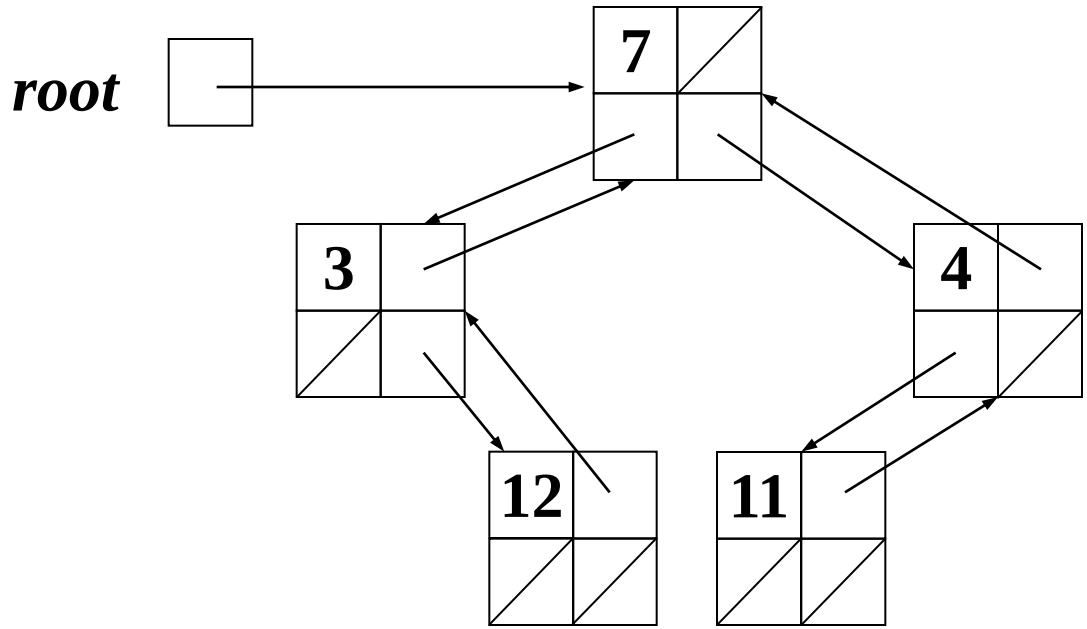
7	1
3	2
4	3
–	4
12	5
11	6
–	7

Representation of binary trees (II)

- A more efficient representation of binary trees is the *linked representation*. A binary tree is represented as a set of linked nodes. Each node x has a key $key[x]$, a parent link $p[x]$, a left link $left[x]$ and a right link $right[x]$.

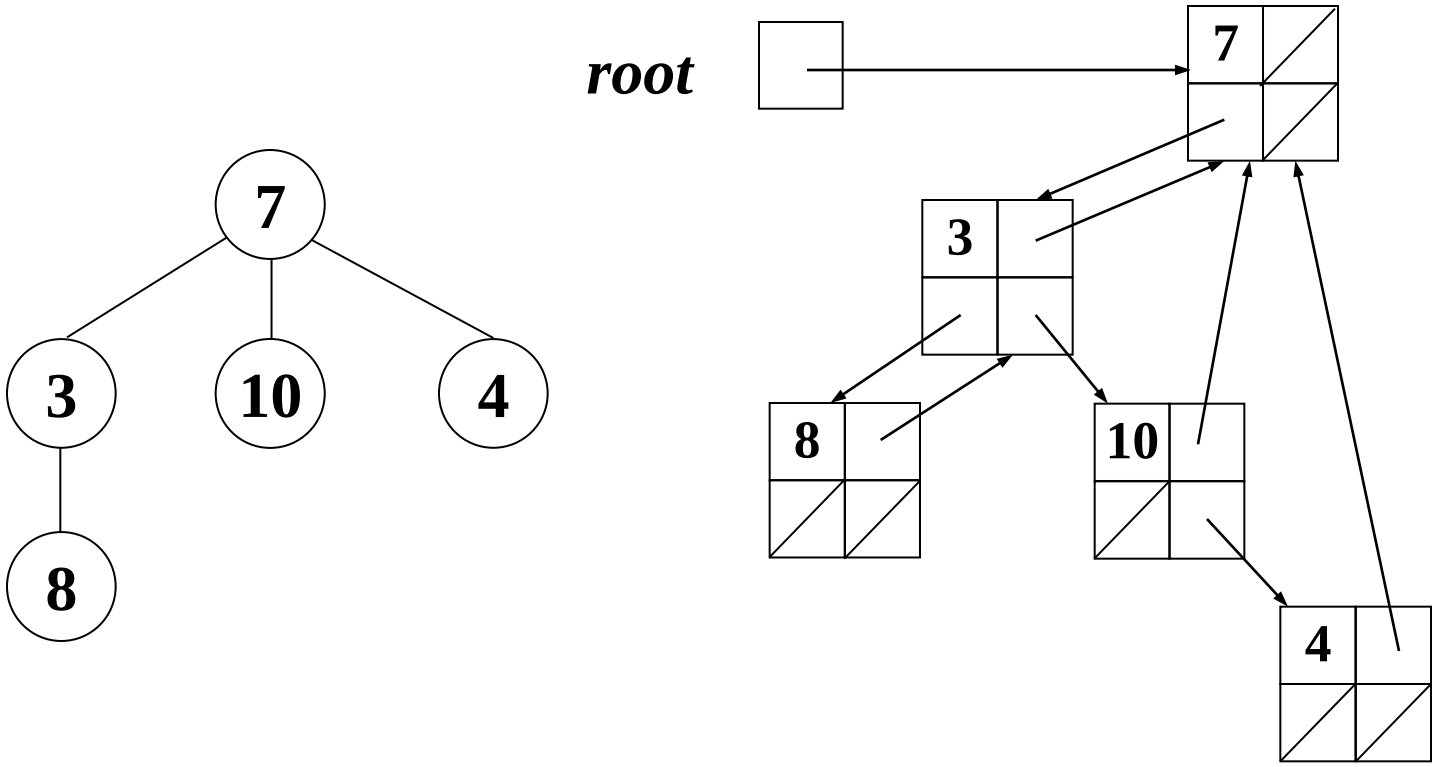
key	***p***
left	***right***

- The root of a tree T is given by a pointer $root[T]$.



Representation of general rooted trees

- If the number of children of each node has an upper bound of k we can replace the fields *left* and *right* with an array *children* of size k. This representation doesn't work if the number of children of a node doesn't have an upper bound or the upper bound isn't known in advance. The representation is inefficient if the tree has many nodes with a number of children significantly less than k.
- Happily, there is an efficient representation of general rooted trees as binary trees which is called the *child-sibling representation* for obvious reasons. Each node x has a parent link $p[x]$, a first-child link *first-child*$[x]$ and a next-sibling link *next-sibling*$[x]$.



Binary tree traversal

- Very often we might need to *traverse* a tree, i.e. visit each node in the tree exactly once. A full traversal produces a linear ordering for the information in the tree.
- When traversing a binary tree we want to treat each node and its sub-trees in the same fashion. If we let L, D, R stand for moving left, visiting the root (data) and moving right the there are six possible traversals: LDR, LRD, DLR, DRL, RDL and RLD. If we adopt the convention that we traverse left before right then only three traversals remain: LDR, LRD and DLR. To these we assign the names: *inorder*, *postorder* and *preorder*.
- A recursive algorithm for inorder traversal is shown below. The algorithms for preorder and postorder traversal are similar.

```
BIN-TREE-INORDER(t)
1. if t ≠ NIL then
2.     BIN-TREE-INORDER(left[t])
3.     VISIT(t)
4.     BIN-TREE-INORDER(right[t])
```

Time complexity of binary tree traversal

- Theorem: If the time required to visit a node is $\Theta(1)$ then the time required to traverse a binary tree with n nodes is $\Theta(n)$.

  Proof:

  Let $T(n)$ be the time required to traverse a binary tree with n nodes.

  We assume that $T(0) = b$ and that the time required to visit a node is a.

  If $n > 0$ and the left sub-tree has m nodes then:

  $T(n) = T(m) + T(n - m - 1) + a$. We shall prove by induction that

  $T(n) = (a+b)n + b$ and the result of the theorem follows trivially.

  The property holds if $n = 0$ because $T(0) = b$.

  We assume that the property holds for $0 \leq m < n$ and we prove that it

  Also holds for $m = n$. According to the recurrence $T(n) = (a + b)m +$

  $b + (a + b)(n - m - 1) + b + a = (a + b)(n - 1) + 2b + a = (a + b)n + b$,

  q.e.d.

Implementation of binary trees – header file

```
#ifndef BINTREE_H
#define BINTREE_H

typedef struct bin_tree_node {
  int key;
  struct bin_tree_node *left,*right;
} BinTreeNode;

typedef struct bin_tree_node *BinTree;

/* constructor */
BinTree binTreeEmpty(void);
BinTree binTree(int k,BinTree l,BinTree r);
/* conversion constructor */
BinTree binTreePInt(int *a);
/* selectors */
BinTree binTreeLeft(BinTree t);
BinTree binTreeRight(BinTree t);
int binTreeKey(BinTree t);
/* tester */
int binTreeIsEmpty(BinTree t);

#endif
```

Implementation of binary trees – c file (I)

```
#include <stdlib.h>
#include "bintree.h"
static BinTree binTreePInt1(int *a,int *i);
BinTree binTreeEmpty(void) {
  return NULL;
}
BinTree binTree(int k,BinTree l,BinTree r) {
  BinTreeNode *t =
    (BinTreeNode *)malloc(sizeof(BinTreeNode));
  t->left = l;
  t->right = r;
  t->key = k;
  return t;
}
BinTree binTreePInt(int *a) {
  int i = 0;
  return binTreePInt1(a,&i);
}
```

Implementation of binary trees – c file (II)

```
static BinTree binTreePInt1(int *a,int *i) {
  if (a[*i] <= 0) {
    (*i)++; return NULL;
  }
  else {
    int k;
    BinTree l,r;
    k = a[(*i)++];
    l = binTreePInt1(a,i);
    r = binTreePInt1(a,i);
    return binTree(k,l,r);
  }
}
BinTree binTreeLeft(BinTree t) {  return t->left; }
BinTree binTreeRight(BinTree t) { return t->right; }
int binTreeKey(BinTree t) { return t->key; }
int binTreeIsEmpty(BinTree t) { return (t==NULL); }
```

Implementation of binary trees – explanation

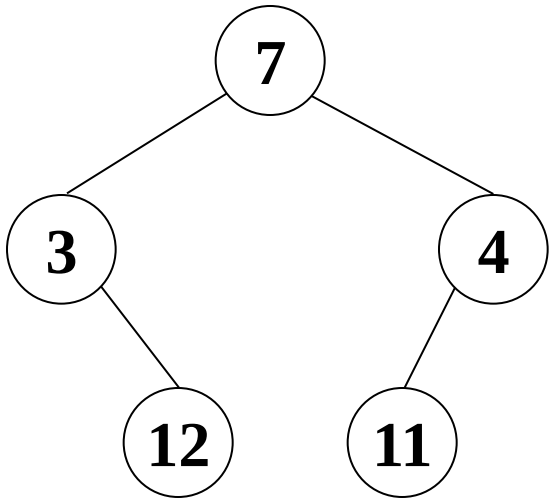
- Implementation of binary trees as an ADT contains the following functions (operations):
  - Constructors:
    - Constructor of an empty binary tree: `binTreeEmpty()`
    - Constructor taking the value of the root, the left sub-tree and the right sub-tree: `binTree()`
    - Constructor that takes the values of the nodes from an array given as a pointer to an integer: `binTreePInt()`
  - Selectors:
    - Selector of the left sub-tree of a nonempty binary tree: `binTreeLeft()`
    - Selector of the right sub-tree of a nonempty binary tree: `binTreeRight()`
    - Selector of the root value of a nonempty binary tree: `binTreeKey()`
  - Recognizers (testers):
    - Recognizer of an empty binary tree: `binTreeIsEmpty()`
- Implementation of `binTreePInt()` uses a helper function `binTreePInt1()` that constructs a binary tree with values taken from an array of integers, starting from a given index *i*. *i* is incremented as the construction progresses. Note that the helper function is not exported outside the module, i.e. is "private" to the module. Therefore, it is declared as `static`.

Main program

- The main program reads a sequence of lines of text, each line describing a binary tree. The keys are strictly positive integers. A negative or zero value indicates a missing information.
- A line contains: the key of the root followed by the keys in the left sub-tree then followed by the keys in the right sub-tree. For example the tree shown below is given as:

  7 3 0 12 0 0 4 11 0 0 0
- The programs stops after reading an empty tree.
- For each non-empty tree the program displays the tree in a structured fashion:

```
        4
            11
    7
            12
        3
```



Example of input – output

Input:

```
2 0 0
2 3 0 0 4 0 0
2 3 4 10 0 0 11 0 0 5 0 0 6 7 15 0 0 12 0 0 8 9 0 0 0
0
```

Output:

```
 2
-------------
      4
  2
      3
-------------
          8
              9
      6
              12
          7
              15
  2
          5
      3
              11
          4
              10
-------------
```

C code of the main program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "bintree.h"
void printTree(BinTree t,int level) {
  if (!binTreeIsEmpty(t)) {
    int i;
    printTree(binTreeRight(t),level+1);
    for (i=0;i<3*level;i++) {
      printf(" ");
    }
    printf("%3d\n",binTreeKey(t));
    printTree(binTreeLeft(t),level+1);
  }
}
void stringToArrayOfInt(char *s,
  int *a) {
  char seps[] = " ";
  const char *token;
  int j = -1;
  token = strtok(s,seps);
  while (token != NULL) {
    a[++j] = atoi(token);
    token = strtok(NULL,seps);
  }
}
char line[100];
int a[50];
BinTree t;
void main() {
  gets(line);
  stringToArrayOfInt(line,a);
  t = binTreePInt(a);
  while (!binTreeIsEmpty(t)) {
    printTree(t,0);
    puts("--------------");
    gets(line);
    stringToArrayOfInt(line,a);
    t = binTreePInt(a);
  }
}
```

Problems

1. Let us consider a binary tree such that each tree node contains a distinct natural number.

   a. Prove that if we know the two sequences of values representing the inorder and the postorder (preorder) of the tree, than the tree is uniquely determined. Design and implement an algorithm for constructing a binary tree based on its inorder and postorder (preorder) sequences. What is the complexity of your algorithm for a binary tree with *n* nodes?

   b. Does this property hold if we know the preorder and the postorder sequences of the binary tree? Argument your answer.

2. Design and implement iterative algorithms for the preorder, postorder and inorder travesals of binary trees.

Problems (continuation)

3. Design and implement an algorithm for computing the set of leafs of a binary tree. What is the complexity of your algorithm?
4. Design and implement an algorithm for computing the height of a binary tree. What is the complexity of your algorithm?
5. A weighted binary tree is a binary tree for which each node n has assigned a positive weight $w(n) > 0$. The *total weight* of a leaf node l is defined as the sum of the weights of each node of the unique path from root r to node l, including r and l. Design and implement an algorithm to determine the leaf node of maximum total weight of the tree. What is the complexity of your algorithm?