POSTGRADUATE DEPARTMENT OF COMPUTER APPLICATIONS,
GOVERNMENT ARTS COLLEGE(AUTONOMOUS),
COIMBATORE 641018.

DATA STRUCTURES AND ALGORITHMS

The contents in this E material are from

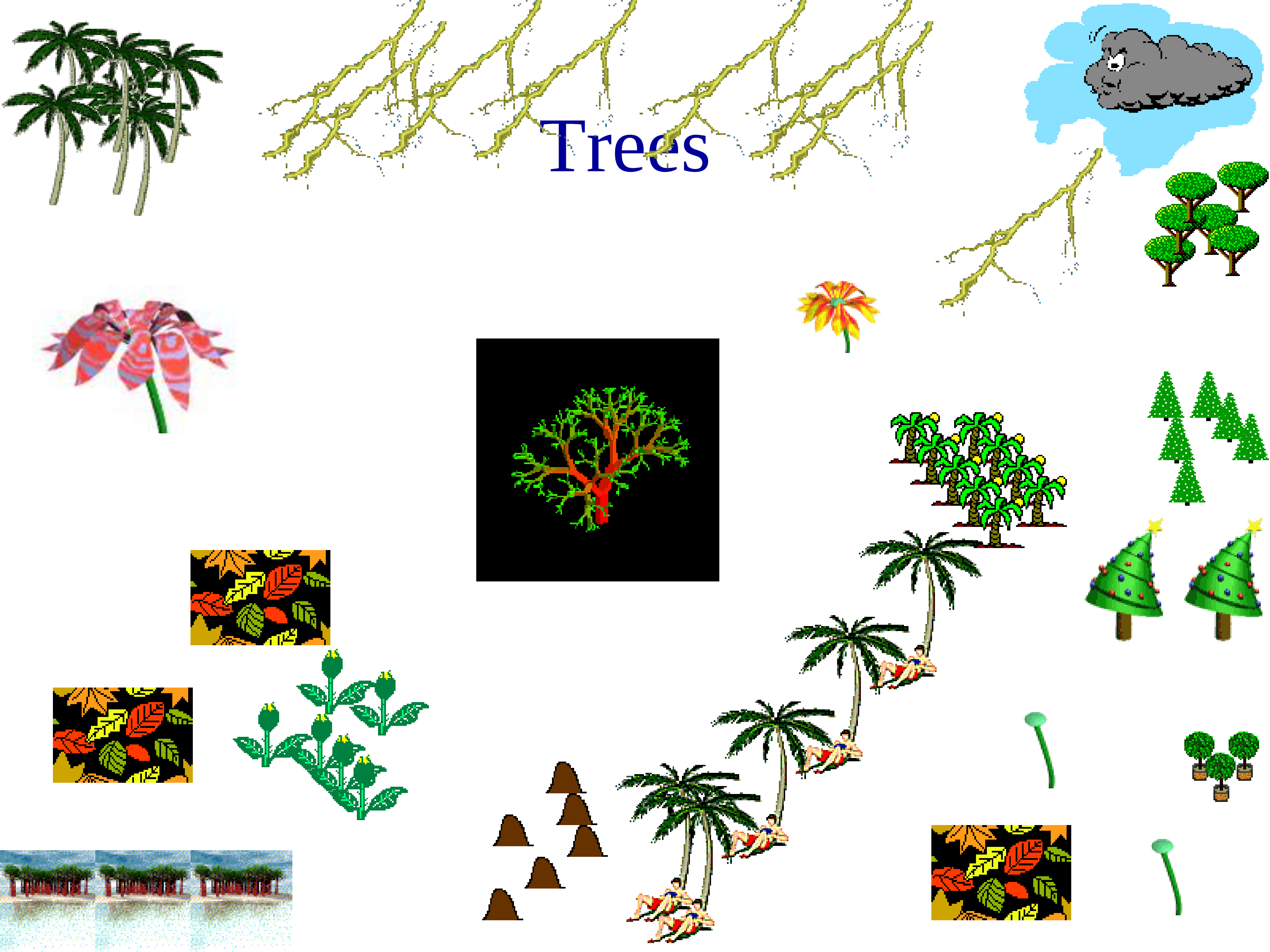
Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
“Fundamentals of Data Structures in C”,
Computer Science Press, 1992.

UNIT 3

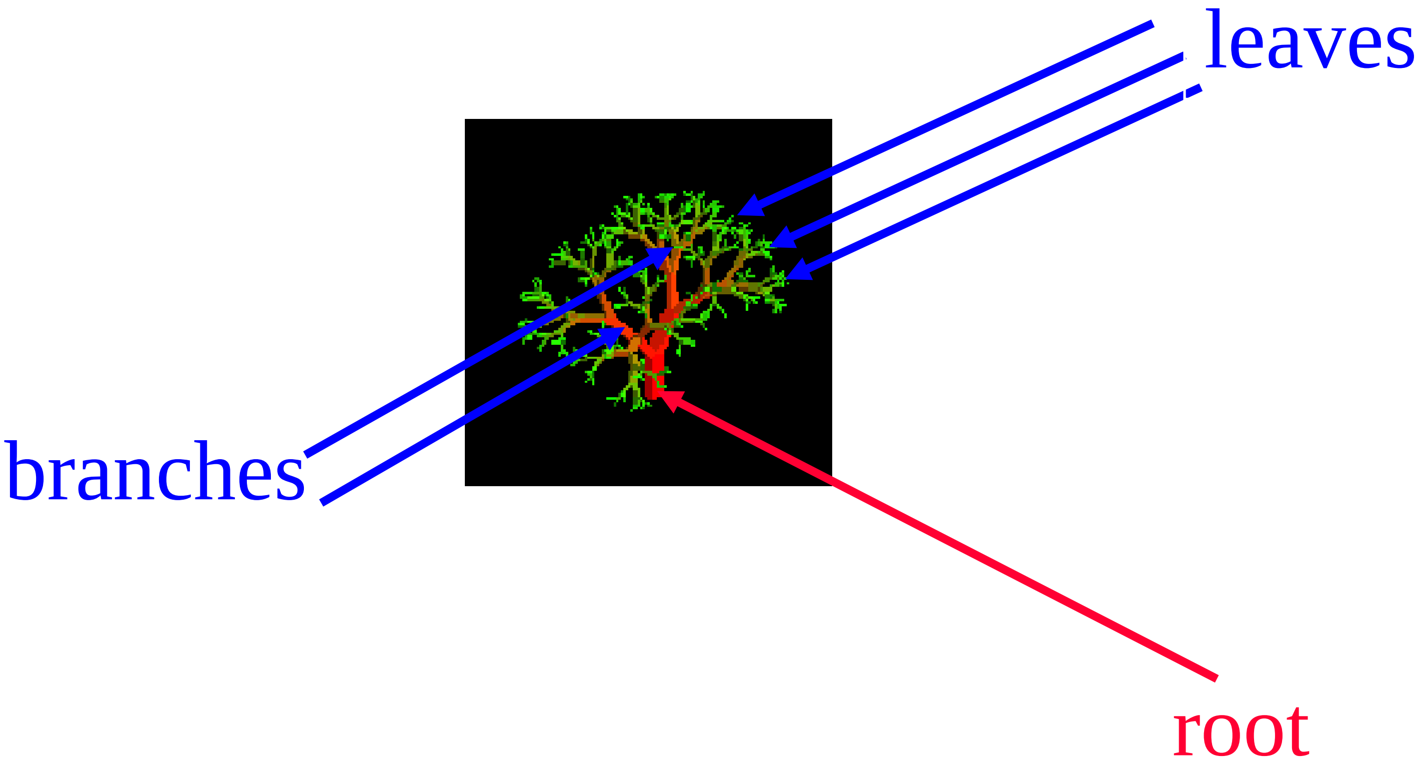
FACULTY

Dr.R.A.ROSELINE M.Sc.M.Phil.,Ph.D,
Associate Professor and Head,
Postgraduate Department of Computer Applications,
Government Arts College(Autonomous),
Coimbatore 641018.

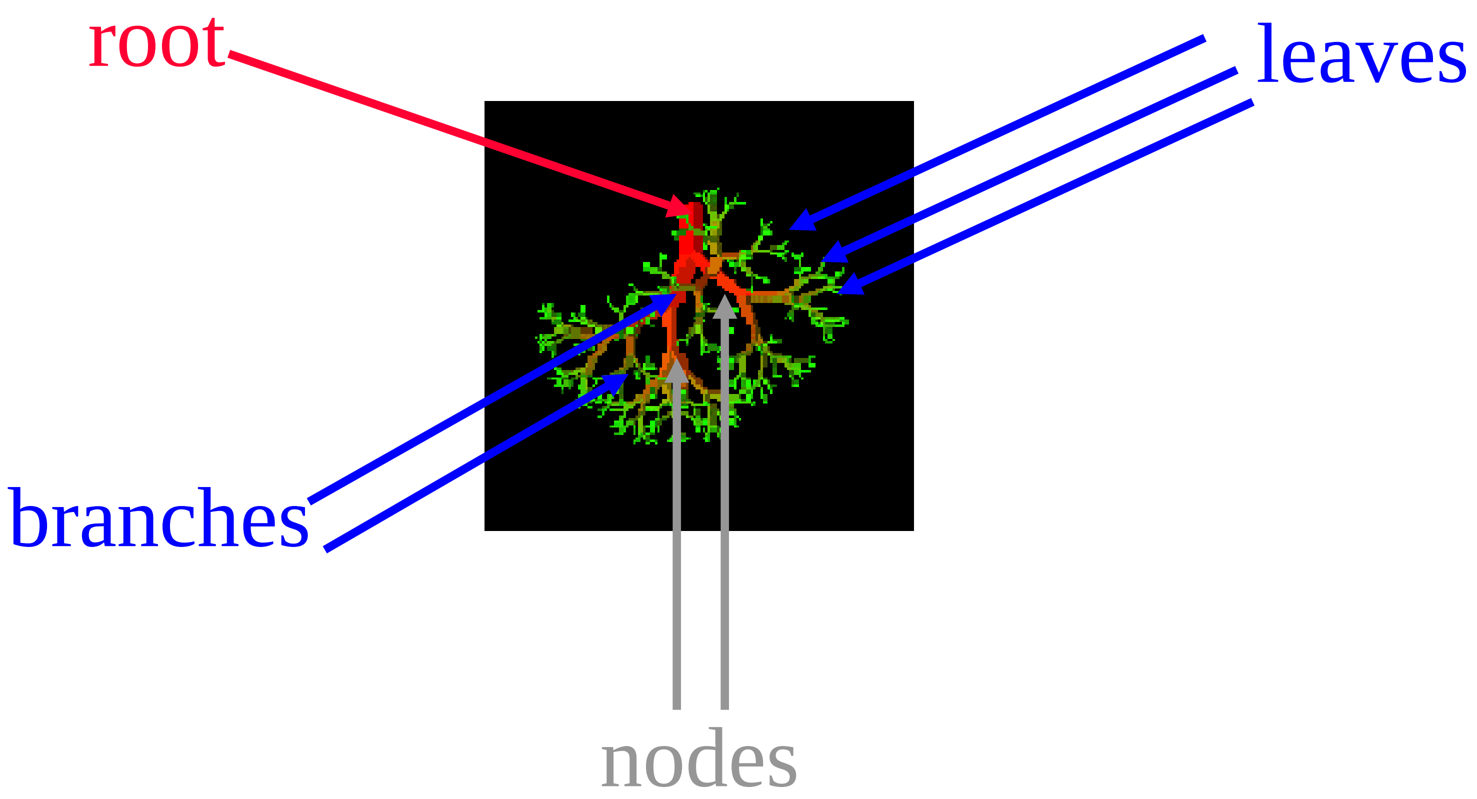
Trees



Nature Lover's View Of A Tree



Computer Scientist's View



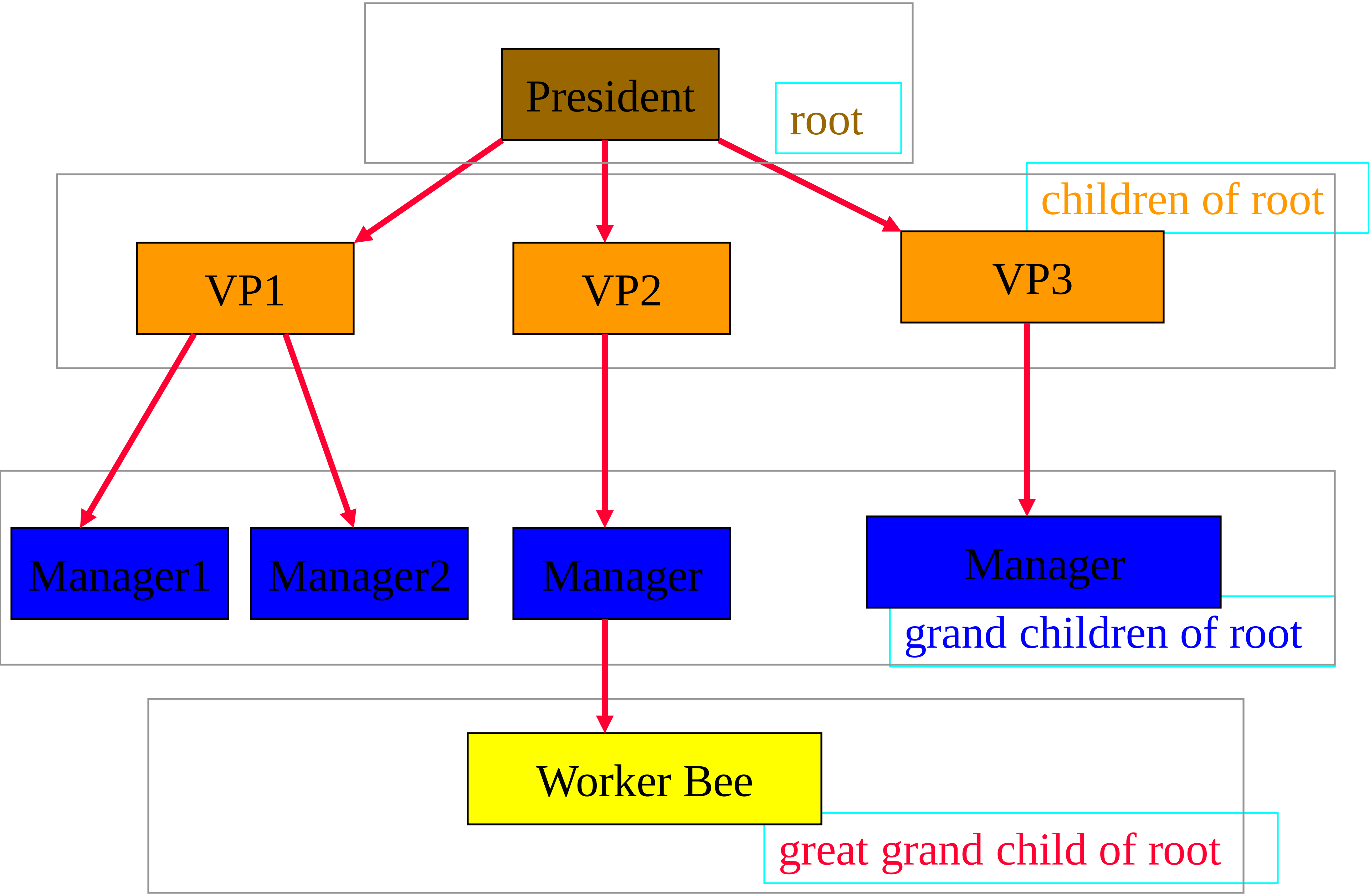
Linear Lists And Trees

- Linear lists are useful for serially ordered data.
  - $(e_0, e_1, e_2, \ldots, e_{n-1})$
  - Days of week.
  - Months in a year.
  - Students in this class.
- Trees are useful for hierarchically ordered data.
  - Employees of a corporation.
    - President, vice presidents, managers, and so on.

Hierarchical Data And Trees

- The element at the top of the hierarchy is the root.
- Elements next in the hierarchy are the children of the root.
- Elements next in the hierarchy are the grandchildren of the root, and so on.
- Elements that have no children are leaves.

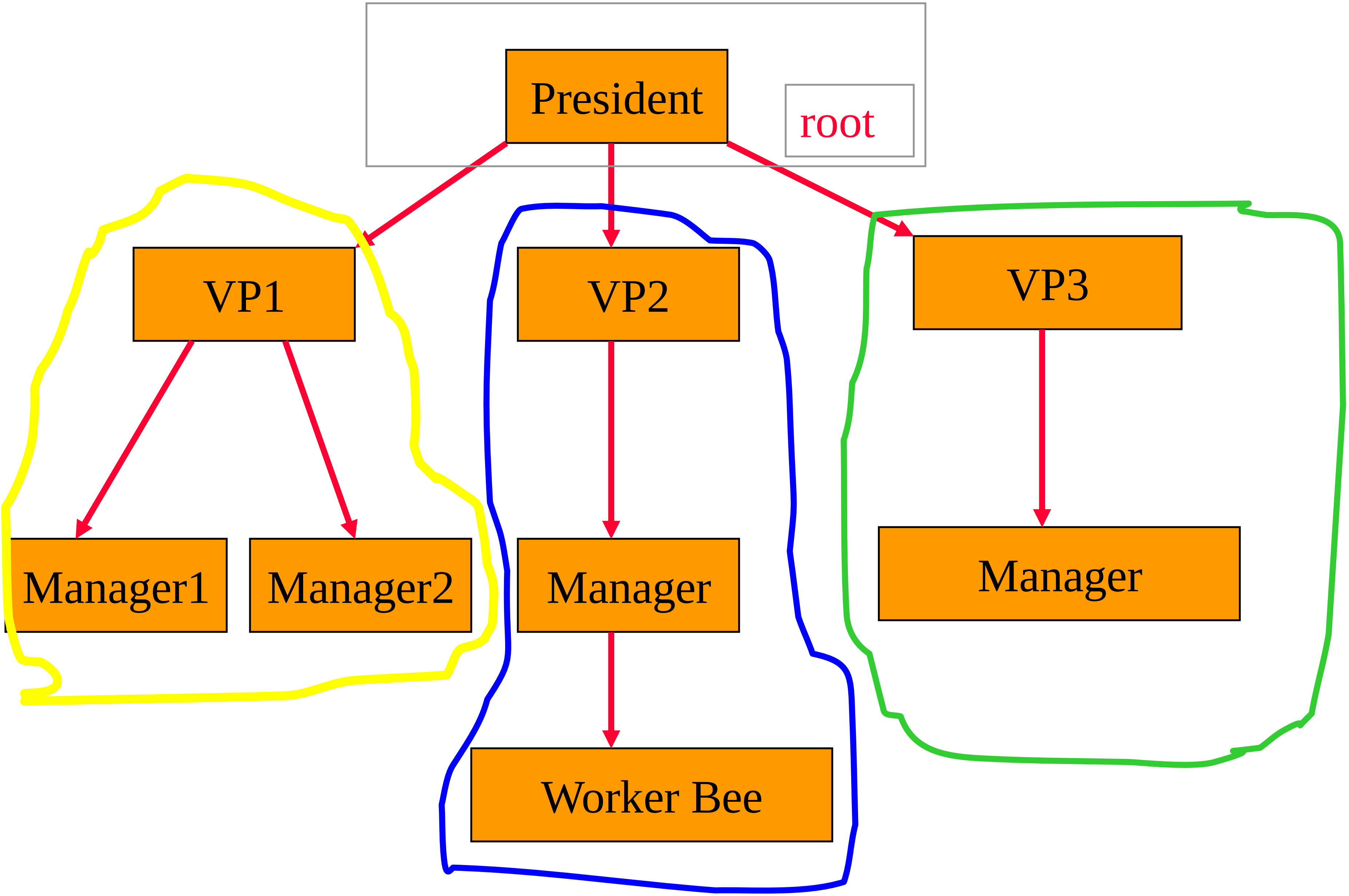
Example Tree



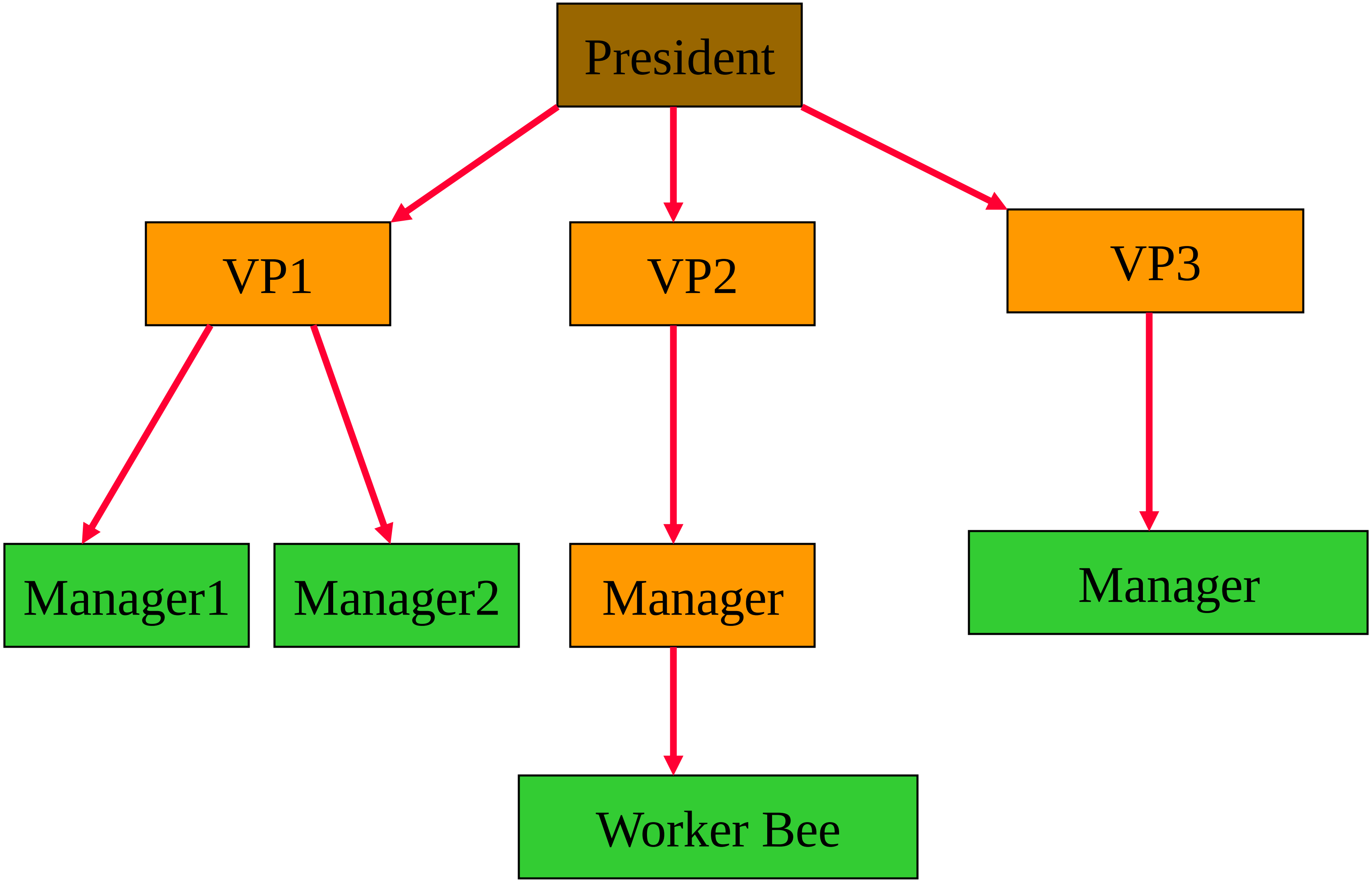
Definition

- A tree t is a finite nonempty set of elements.
- One of these elements is called the root.
- The remaining elements, if any, are partitioned into trees, which are called the subtrees of t.

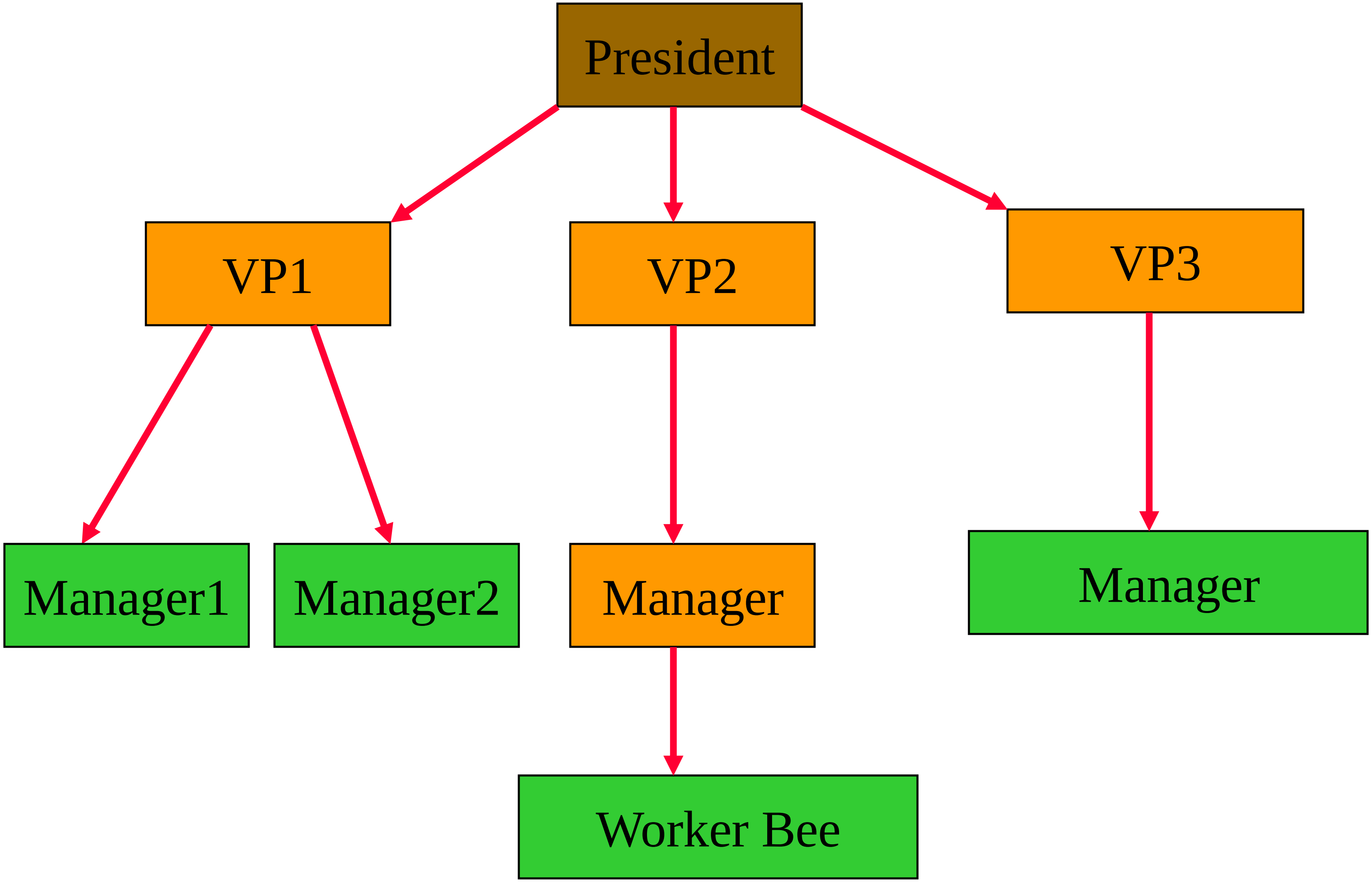
Subtrees


President
root
VP1
VP2
VP3
Manager1
Manager2
Manager
Manager
Worker Bee

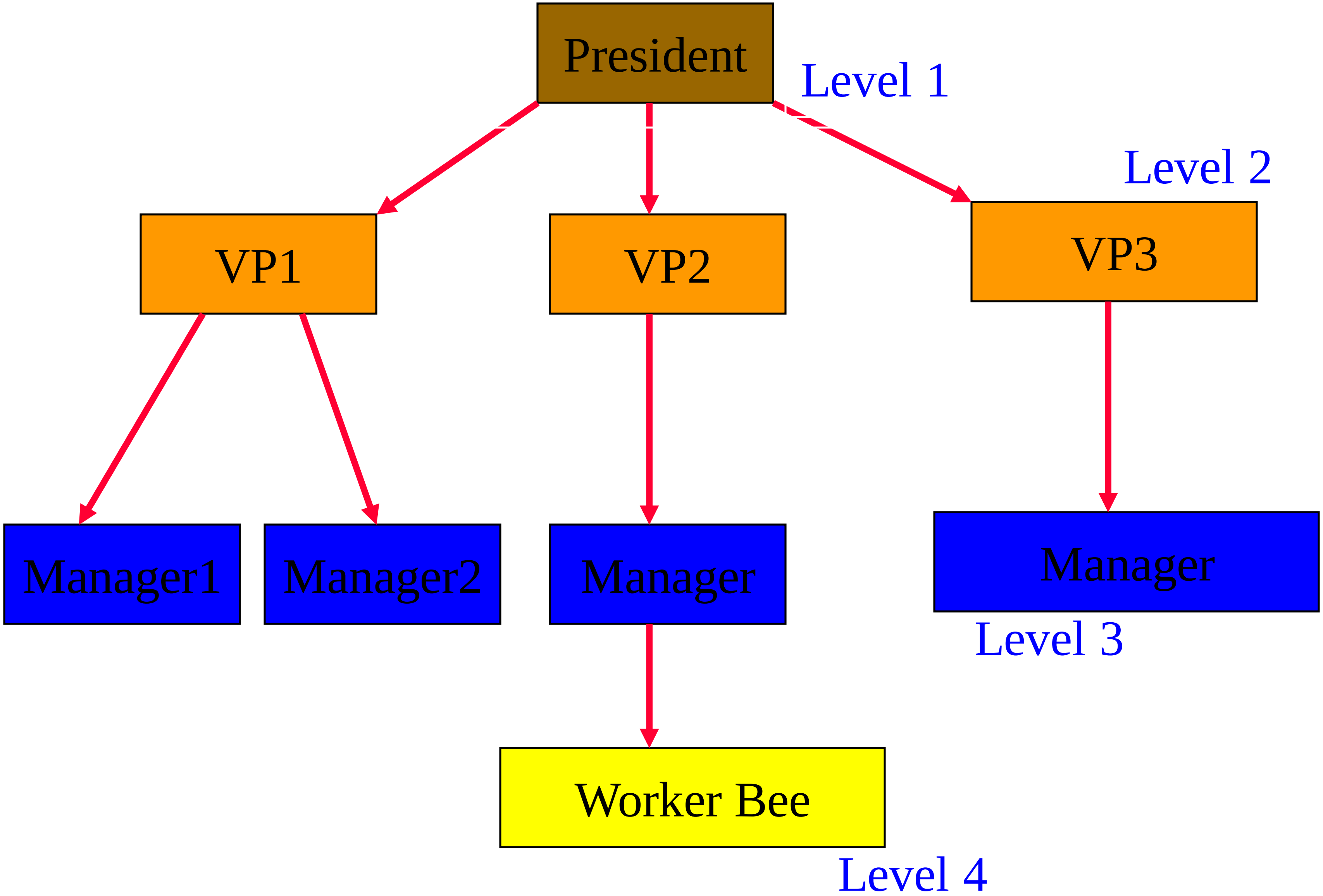
Leaves


President
VP1
VP2
VP3
Manager1
Manager2
Manager
Manager
Worker Bee

Parent, Grandparent, Siblings, Ancestors, Descendants



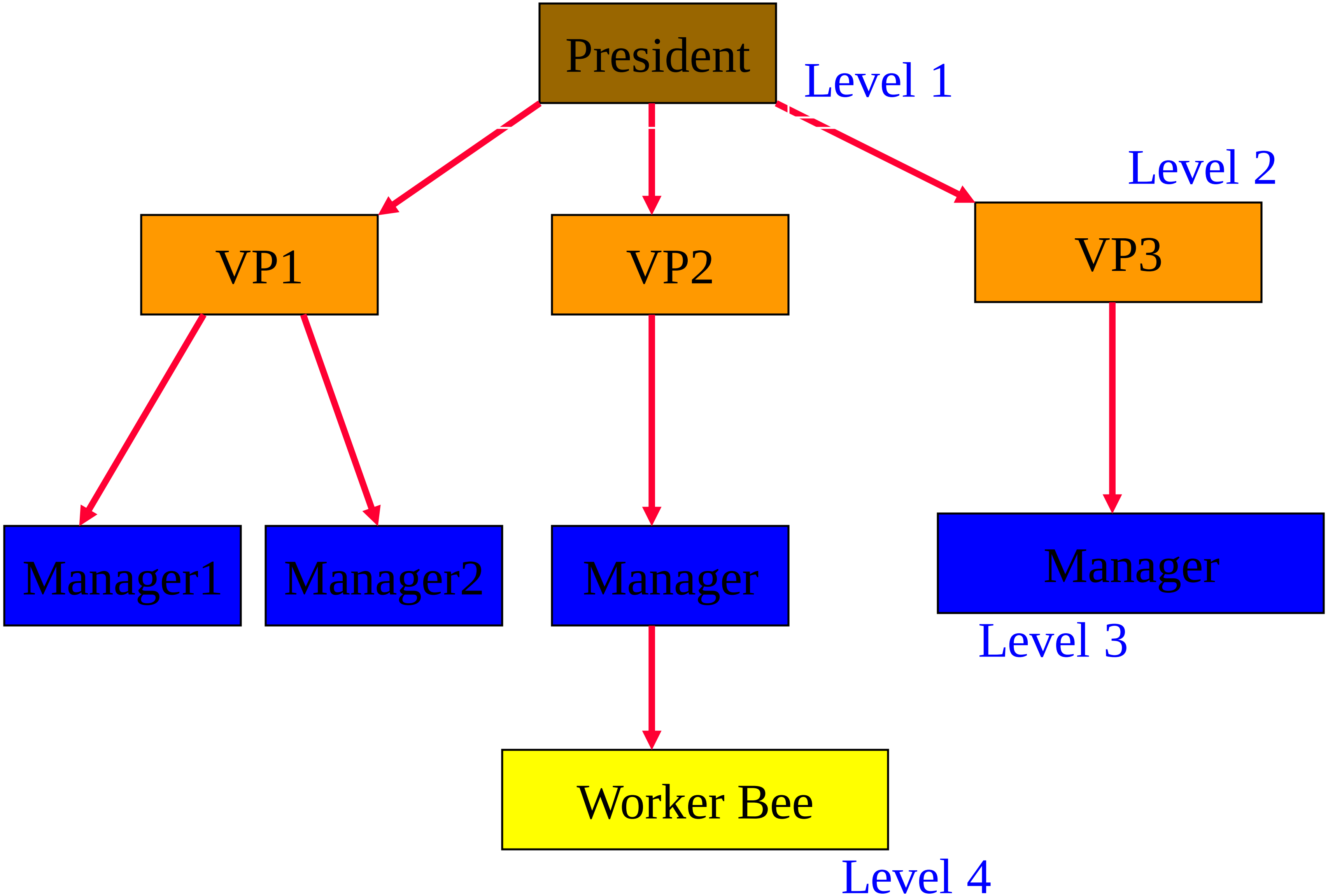
Levels


President
Level 1
Level 2
VP1
VP2
VP3
Manager1
Manager2
Manager
Manager
Level 3
Worker Bee
Level 4

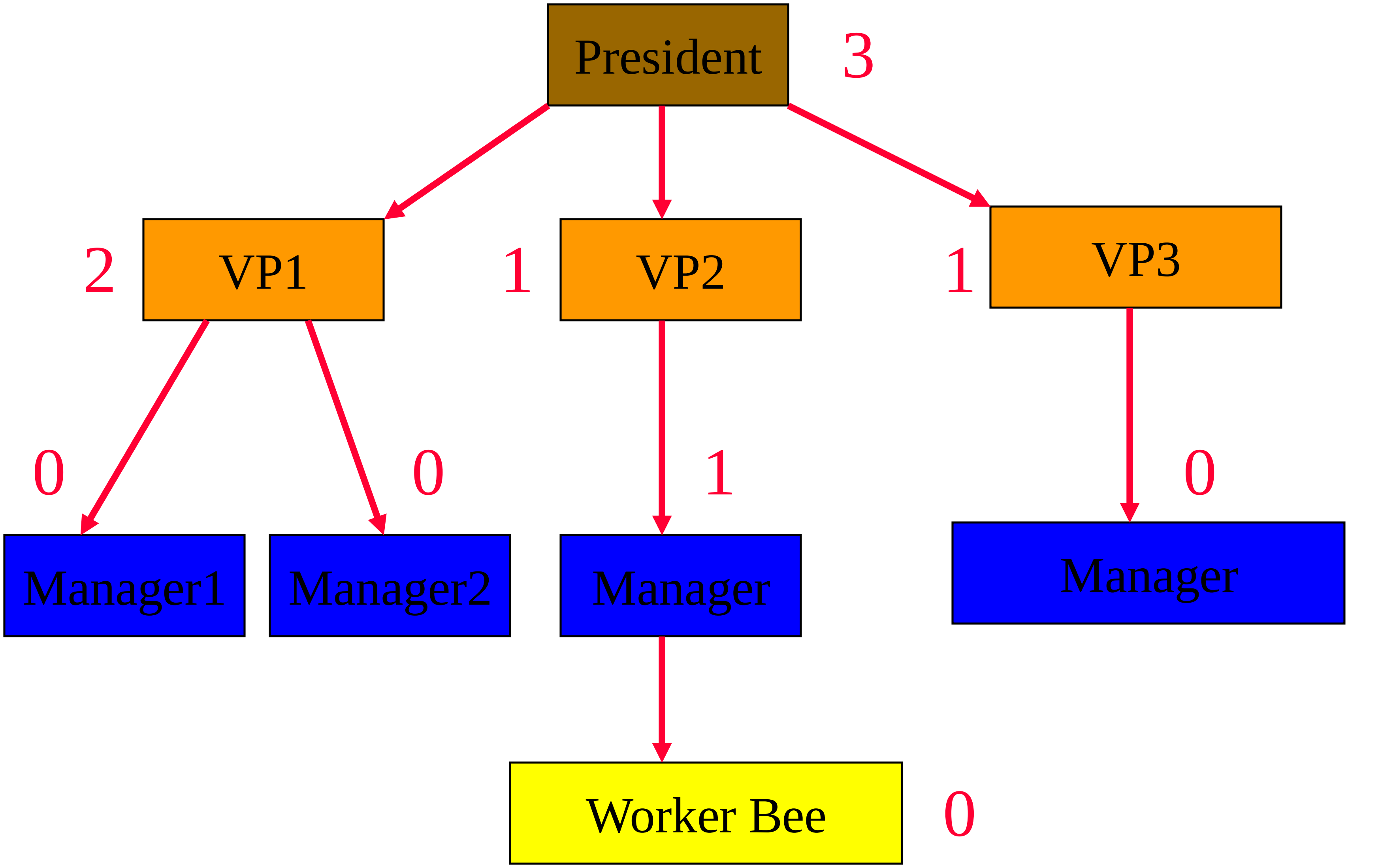
Caution

- Some texts start level numbers at 0 rather than at 1.
- Root is at level 0.
- Its children are at level 1.
- The grand children of the root are at level 2.
- And so on.
- We shall number levels with the root at level 1.

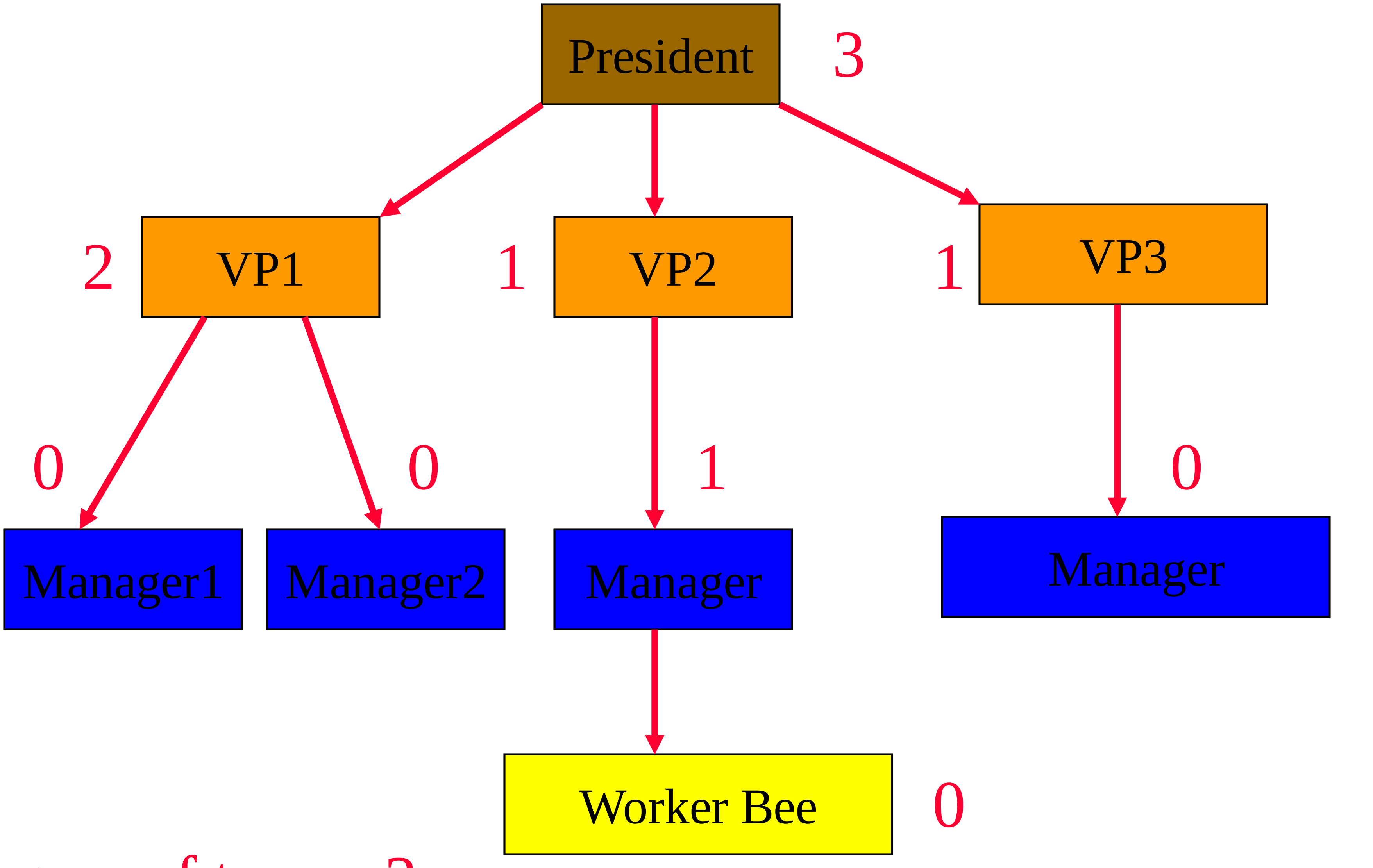
height = depth = number of levels



Node Degree = Number Of Children



Tree Degree = Max Node Degree



Degree of tree = 3.

Binary Tree

- Finite (possibly empty) collection of elements.
- A nonempty binary tree has a root element.
- The remaining elements (if any) are partitioned into two binary trees.
- These are called the left and right subtrees of the binary tree.

Differences Between A Tree & A Binary Tree

- No node in a binary tree may have a degree more than 2, whereas there is no limit on the degree of a node in a tree.
- A binary tree may be empty; a tree cannot be empty.

Differences Between A Tree & A Binary Tree

- The subtrees of a binary tree are ordered; those of a tree are not ordered.

- Are different when viewed as binary trees.
- Are the same when viewed as trees.

Arithmetic Expressions

- (a + b) * (c + d) + e – f/g*h + 3.25
- Expressions comprise three kinds of entities.
  - Operators (+, -, /, *).
  - Operands (a, b, c, d, e, f, g, h, 3.25, (a + b), (c + d), etc.).
  - Delimiters ((,)).

Operator Degree

- Number of operands that the operator requires.
- Binary operator requires two operands.
  - a + b
  - c / d
  - e - f
- Unary operator requires one operand.
  - + g
  - - h

Infix Form

- Normal way to write an expression.
- Binary operators come in between their left and right operands.
    - a * b
    - a + b * c
    - a * b / c
    - (a + b) * (c + d) + e – f/g*h + 3.25

Operator Priorities

- How do you figure out the operands of an operator?
  - a + b * c
  - a * b + c / d
- This is done by assigning operator priorities.
  - priority(*) = priority(/) > priority(+) = priority(-)
- When an operand lies between two operators, the operand associates with the operator that has higher priority.

Tie Breaker

- When an operand lies between two operators that have the same priority, the operand associates with the operator on the left.
  - a + b - c
  - a * b / c / d

Delimiters

- Subexpression within delimiters is treated as a single operand, independent from the remainder of the expression.
  - (a + b) * (c – d) / (e – f)

Infix Expression Is Hard To Parse

- Need operator priorities, tie breaker, and delimiters.
- This makes computer evaluation more difficult than is necessary.
- Postfix and prefix expression forms do not rely on operator priorities, a tie breaker, or delimiters.
- So it is easier for a computer to evaluate expressions that are in these forms.

Postfix Form

- The postfix form of a variable or constant is the same as its infix form.
  - a, b, 3.25
- The relative order of operands is the same in infix and postfix forms.
- Operators come immediately after the postfix form of their operands.
  - Infix = a + b
  - Postfix = ab+

Postfix Examples

- Infix = a + b * c
  - Postfix = a b c * +
- Infix = a * b + c
  - Postfix = a b * c +
- Infix = (a + b) * (c – d) / (e + f)
  - Postfix = a b + c d - * e f + /

Unary Operators

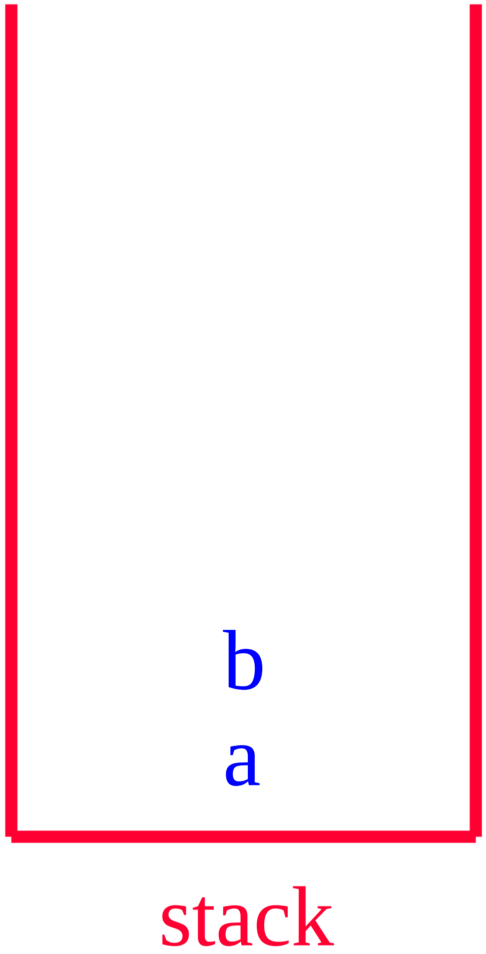
- Replace with new symbols.
  - + a => a @
  - + a + b => a @ b +
  - - a => a ?
  - - a-b => a ? b -

Postfix Evaluation

- Scan postfix expression from left to right pushing operands on to a stack.
- When an operator is encountered, pop as many operands as this operator needs; evaluate the operator; push the result on to the stack.
- This works because, in postfix, operators come immediately after their operands.

Postfix Evaluation

- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /



Postfix Evaluation

- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /

d

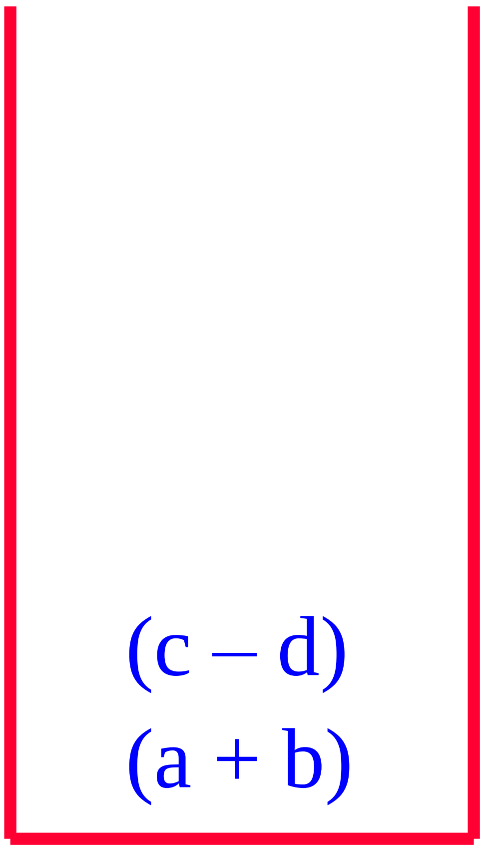
c

(a + b)

stack

Postfix Evaluation

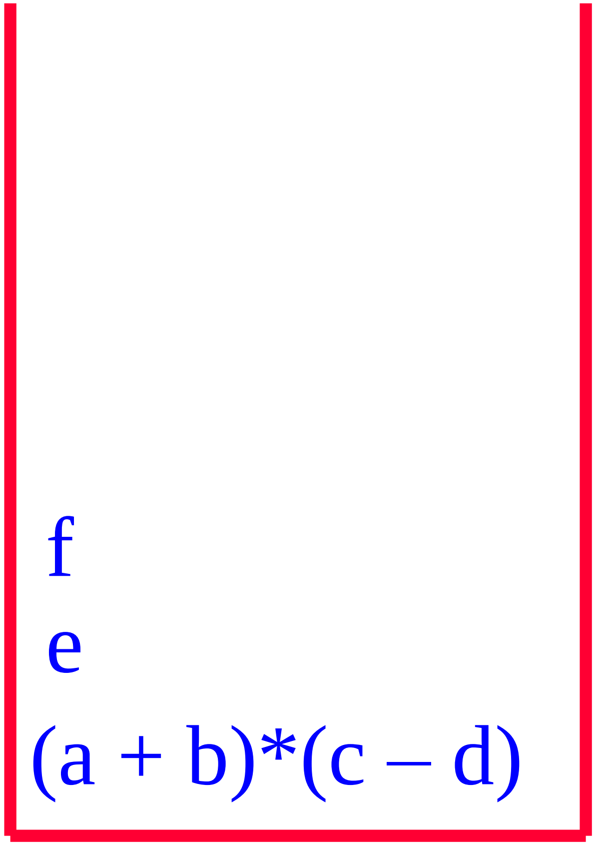
- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /
- a b + c d - * e f + /



stack

Postfix Evaluation

- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /



stack

Postfix Evaluation

- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /

(e + f)

(a + b)*(c – d)

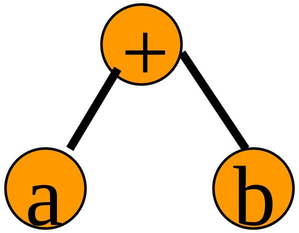
stack

Prefix Form

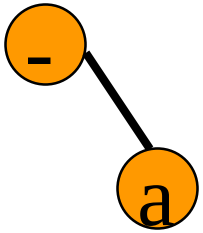
- The prefix form of a variable or constant is the same as its infix form.
  - a, b, 3.25
- The relative order of operands is the same in infix and prefix forms.
- Operators come immediately before the prefix form of their operands.
  - Infix = a + b
  - Postfix = ab+
  - Prefix = +ab

Binary Tree Form

- a + b

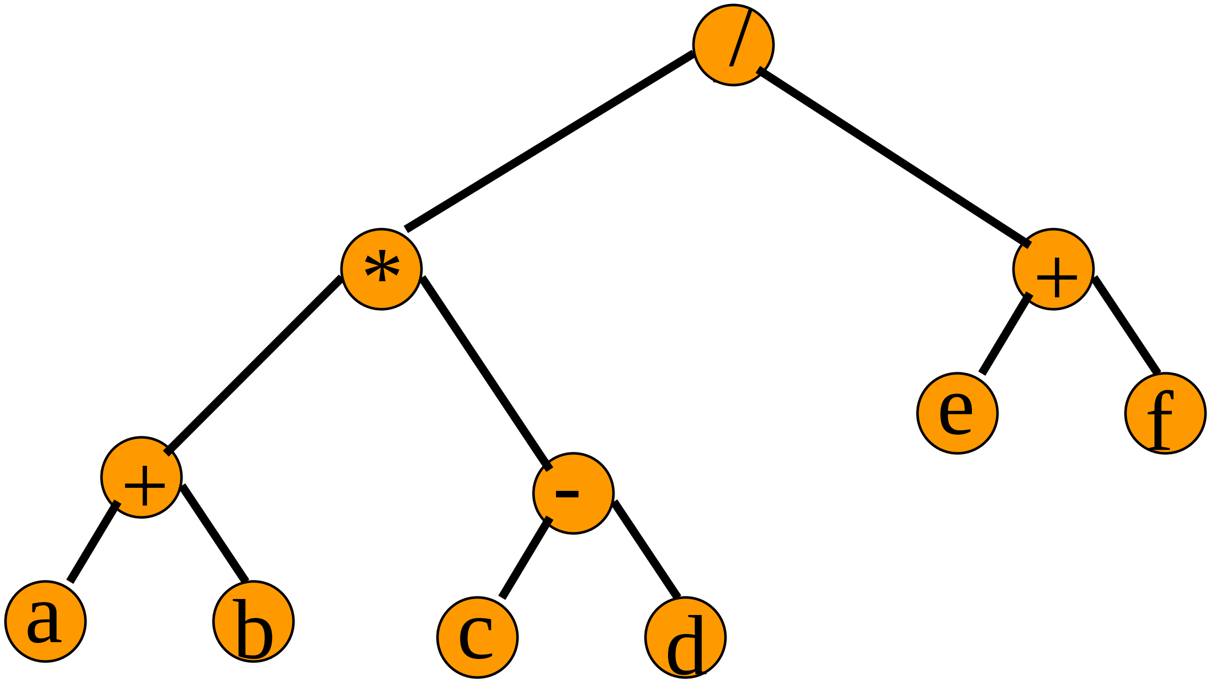


- \- a



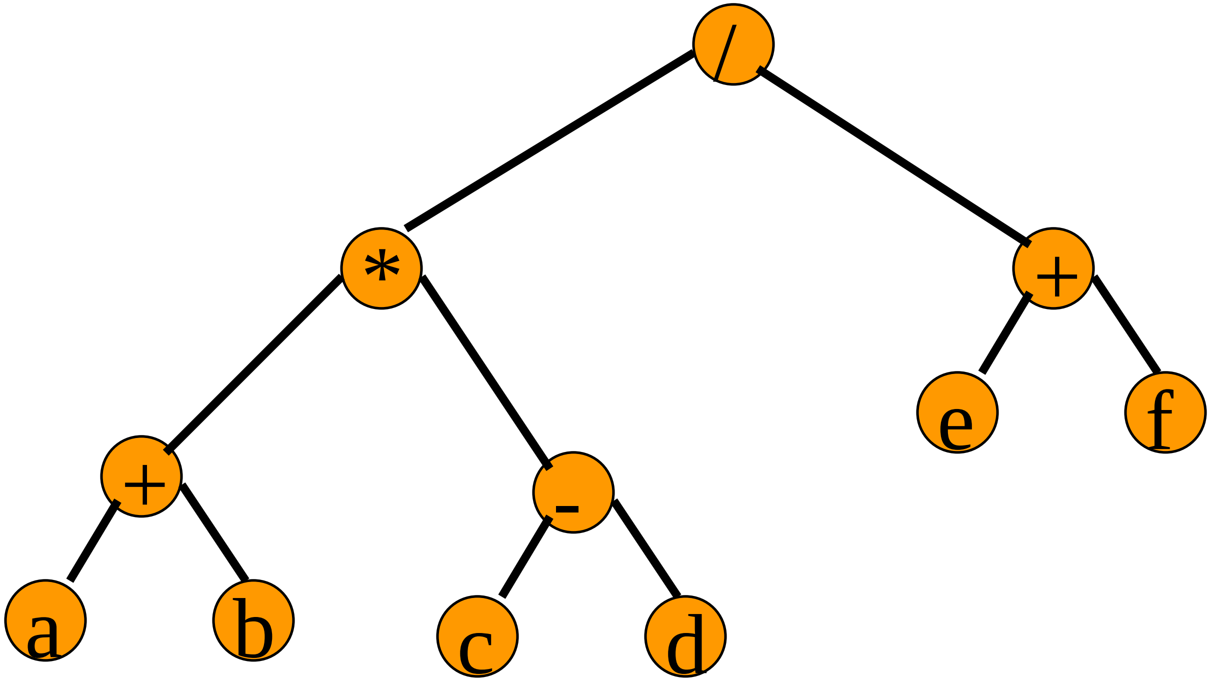
Binary Tree Form

- (a + b) * (c – d) / (e + f)



Merits Of Binary Tree Form

- Left and right operands are easy to visualize.
- Code optimization algorithms work with the binary tree form of an expression.
- Simple recursive evaluation of expression.



Graphs

- G = (V,E)
- V is the vertex set.
- Vertices are also called nodes and points.
- E is the edge set.
- Each edge connects two different vertices.
- Edges are also called arcs and lines.
- Directed edge has an orientation (u,v).

$$u \longrightarrow v$$

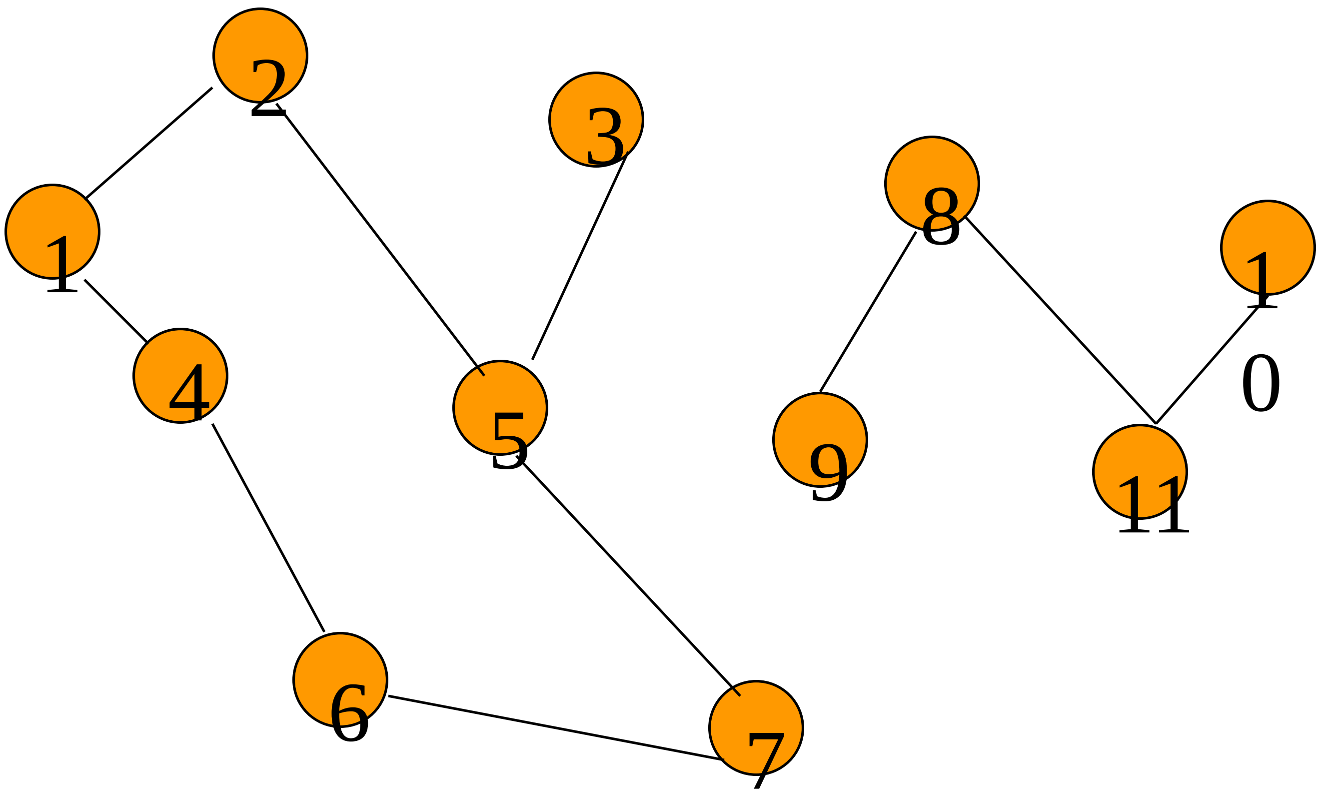
Graphs

- Undirected edge has no orientation (u,v).

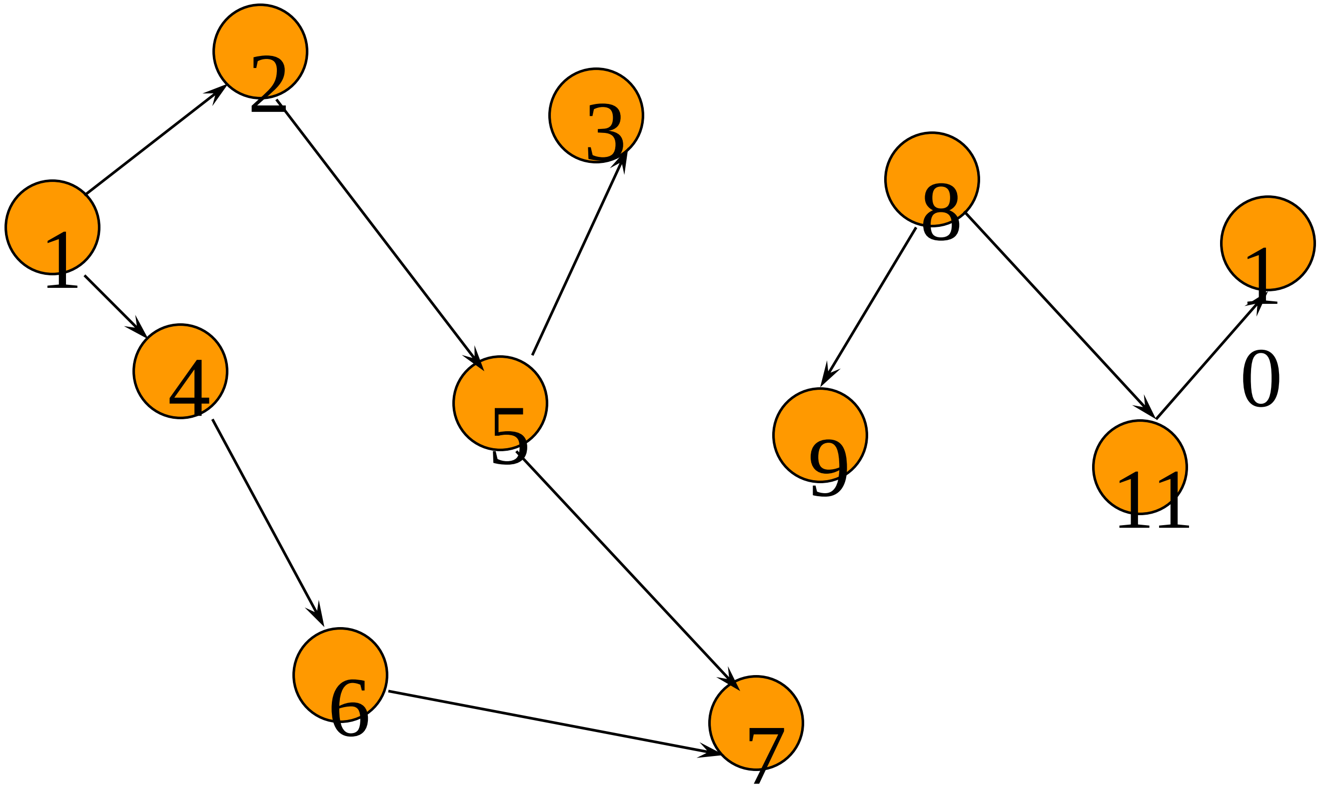
  u ——— v

- Undirected graph => no oriented edge.
- Directed graph => every edge has an orientation.

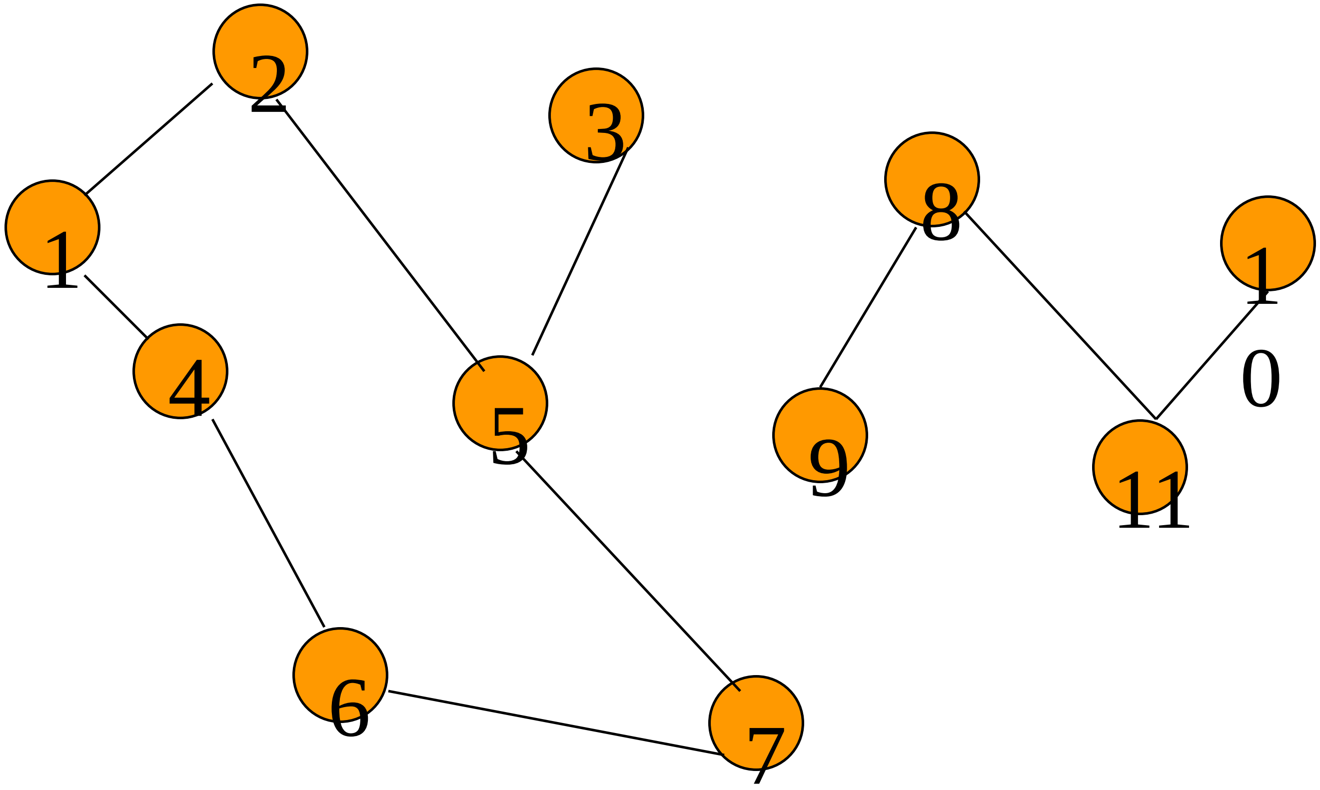
Undirected Graph



Directed Graph (Digraph)

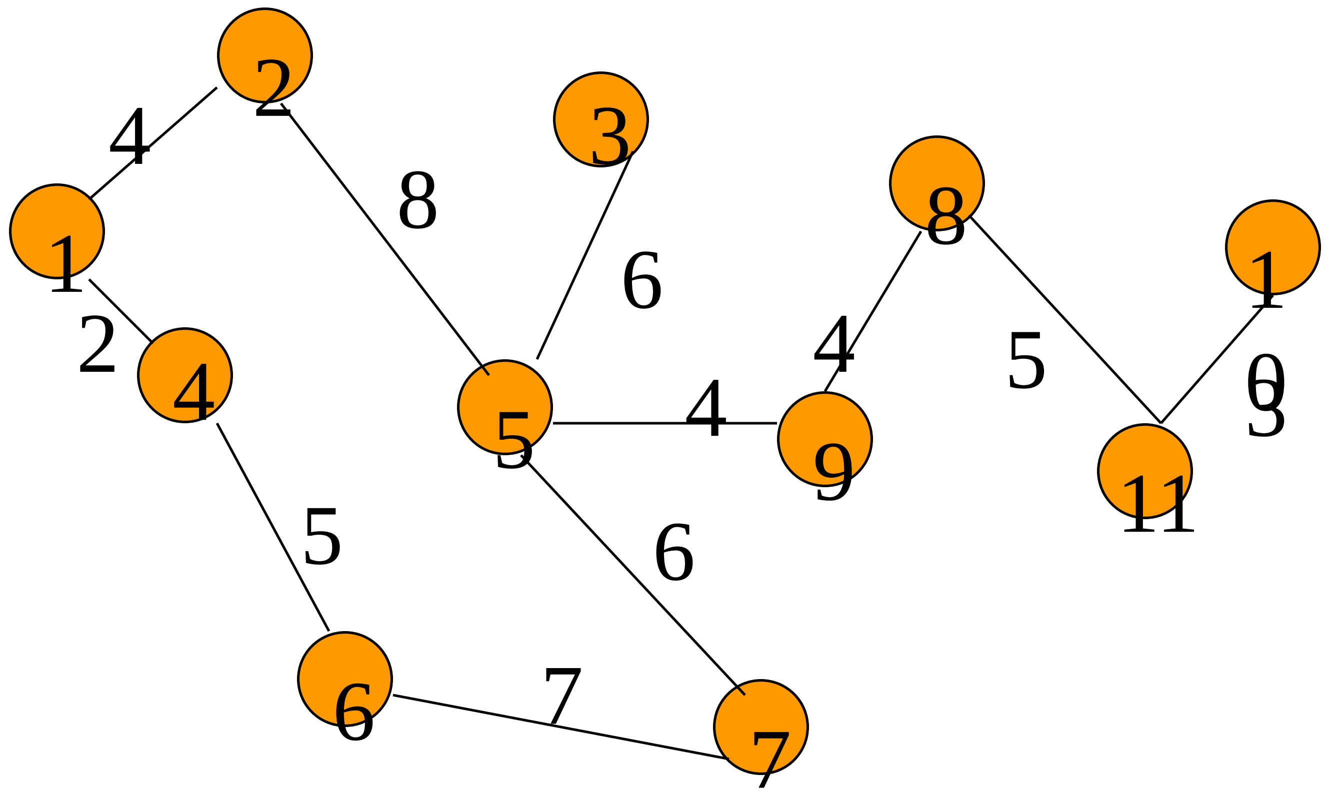


Applications—Communication Network



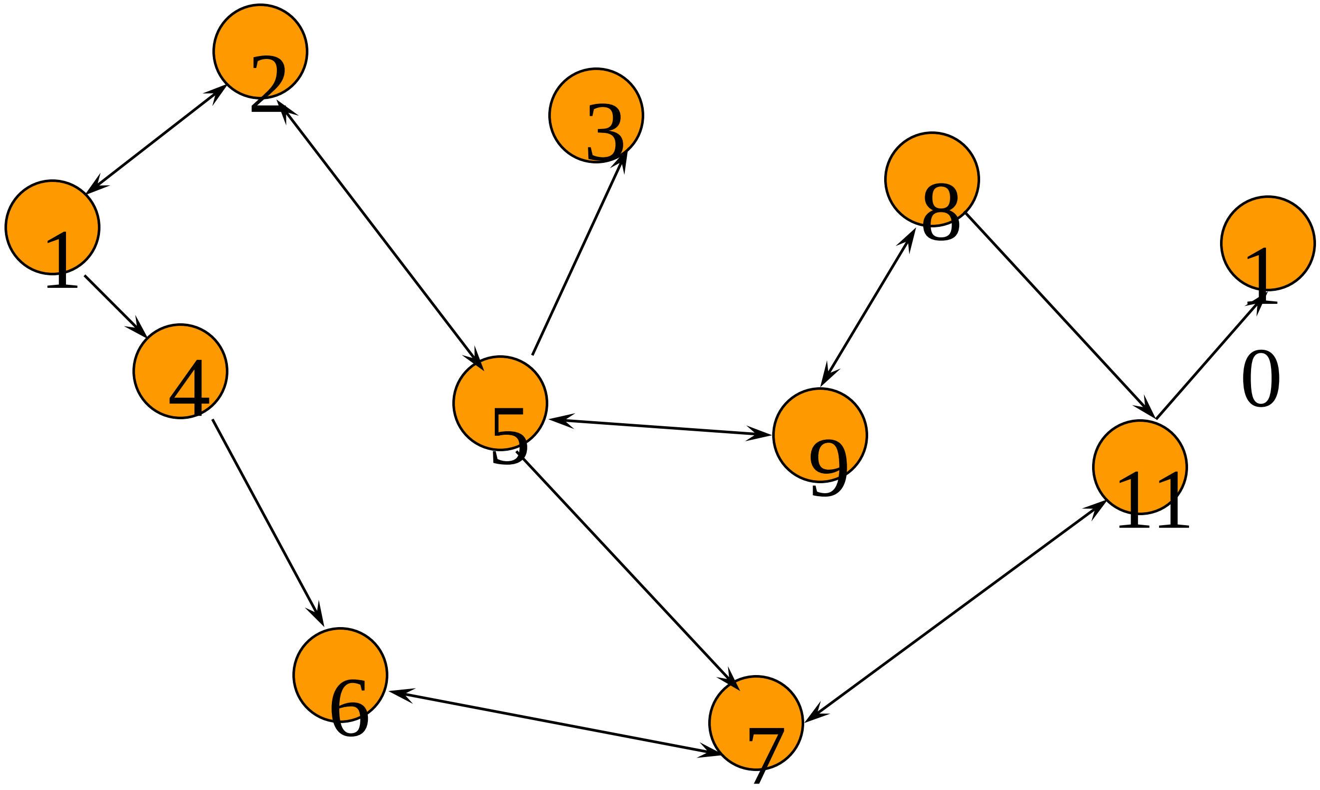
- Vertex = city, edge = communication link.

Driving Distance/Time Map



- Vertex = city, edge weight = driving distance/time.

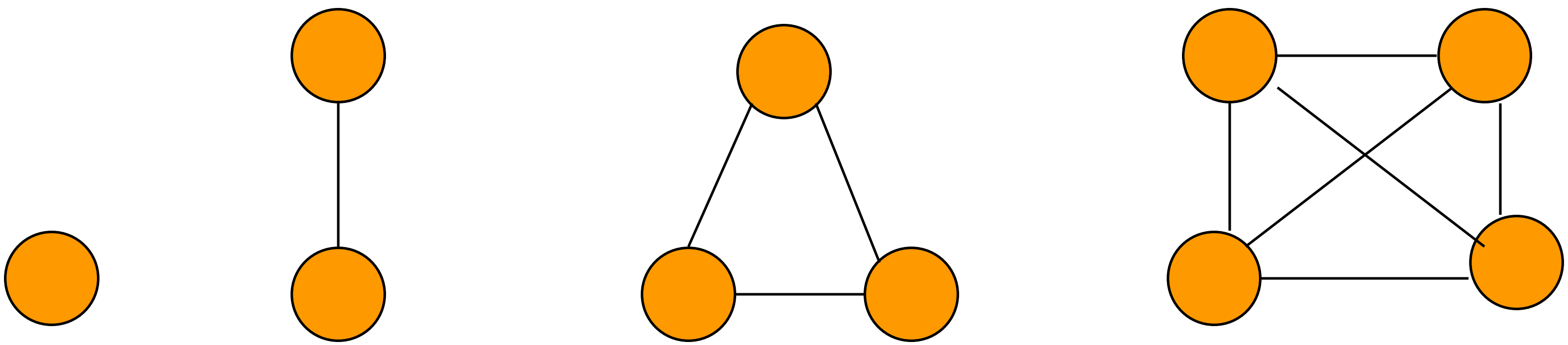
Street Map



- Some streets are one way.

Complete Undirected Graph

Has all possible edges.



n = 1　n = 2　n = 3　n = 4

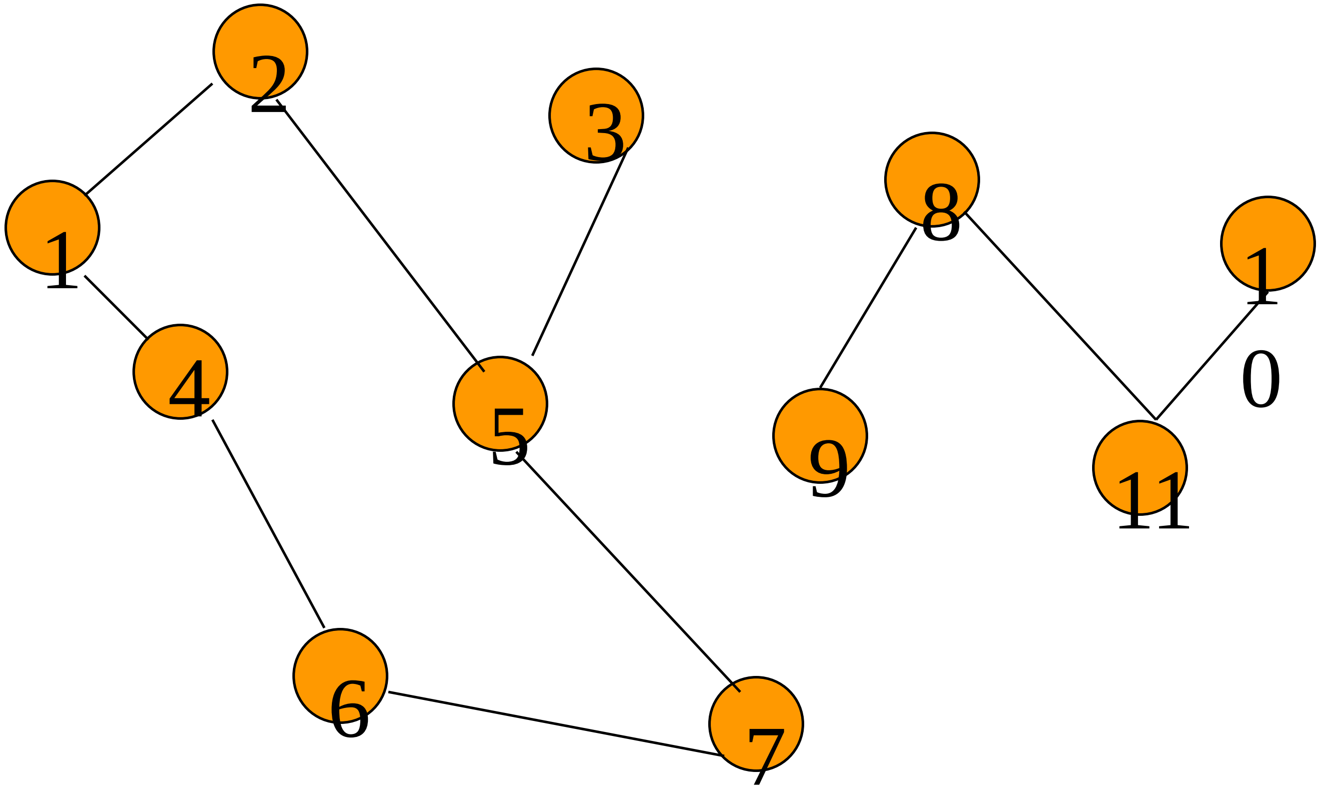
Number Of Edges—Undirected Graph

- Each edge is of the form (u,v), u != v.
- Number of such pairs in an n vertex graph is n(n-1).
- Since edge (u,v) is the same as edge (v,u), the number of edges in a complete undirected graph is n(n-1)/2.
- Number of edges in an undirected graph is <= n(n-1)/2.

Number Of Edges—Directed Graph

- Each edge is of the form (u,v), u != v.
- Number of such pairs in an n vertex graph is n(n-1).
- Since edge (u,v) is not the same as edge (v,u), the number of edges in a complete directed graph is n(n-1).
- Number of edges in a directed graph is <= n(n-1).

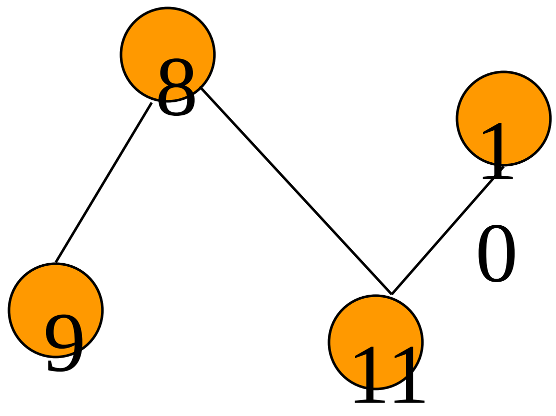
Vertex Degree



Number of edges incident to vertex.

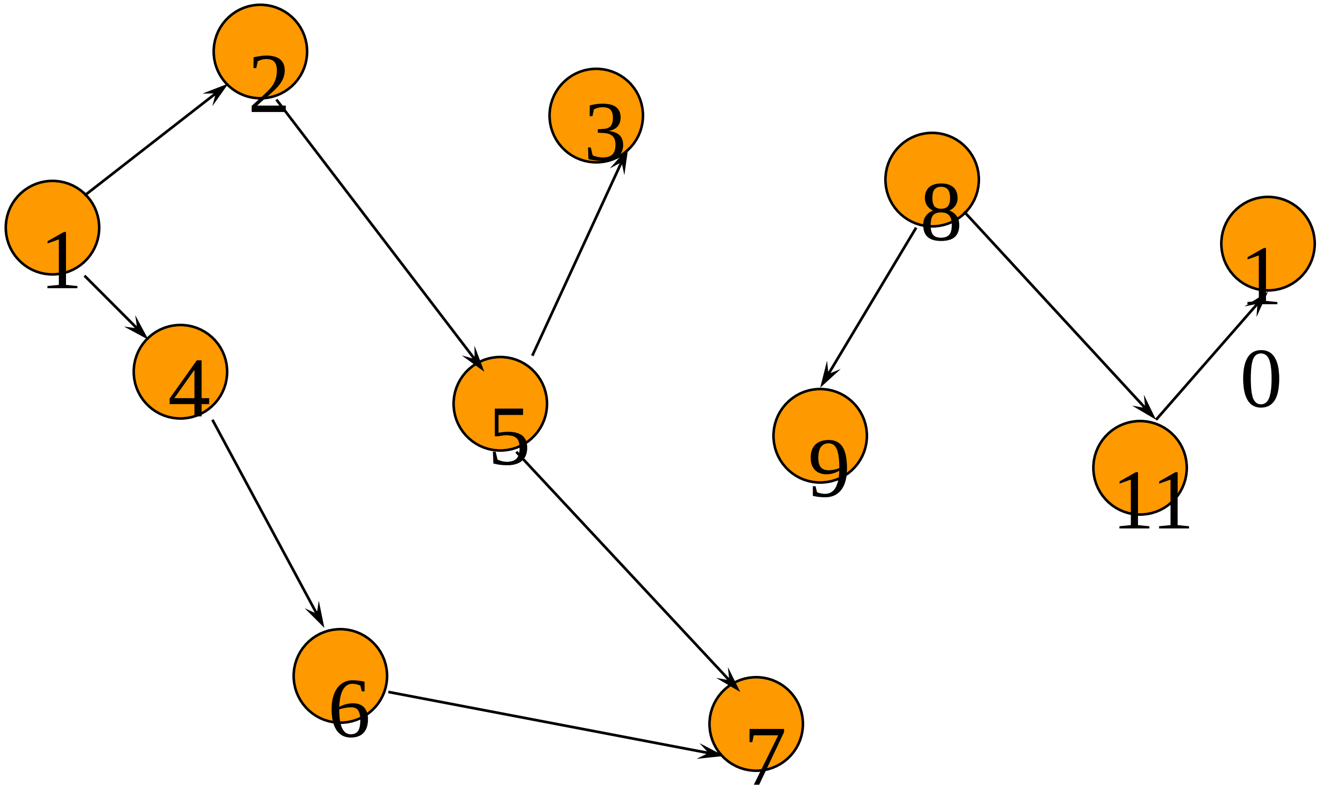
degree(2) = 2, degree(5) = 3, degree(3) = 1

Sum Of Vertex Degrees



Sum of degrees = 2e (e is number of edges)

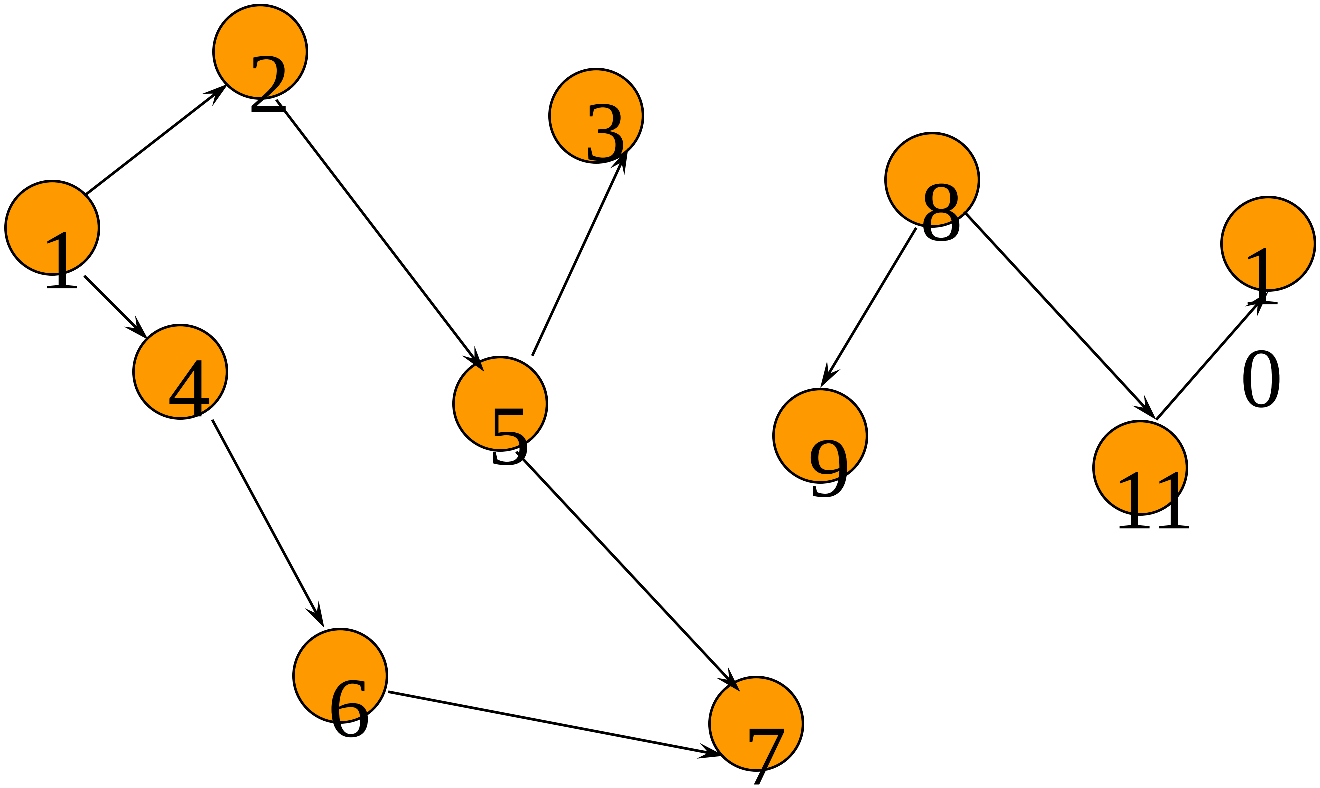
In-Degree Of A Vertex



in-degree is number of incoming edges

indegree(2) = 1, indegree(8) = 0

Out-Degree Of A Vertex



out-degree is number of outbound edges

outdegree(2) = 1, outdegree(8) = 2

Sum Of In- And Out-Degrees

each edge contributes 1 to the in-degree of some vertex and 1 to the out-degree of some other vertex

sum of in-degrees = sum of out-degrees = e,

where e is the number of edges in the digraph

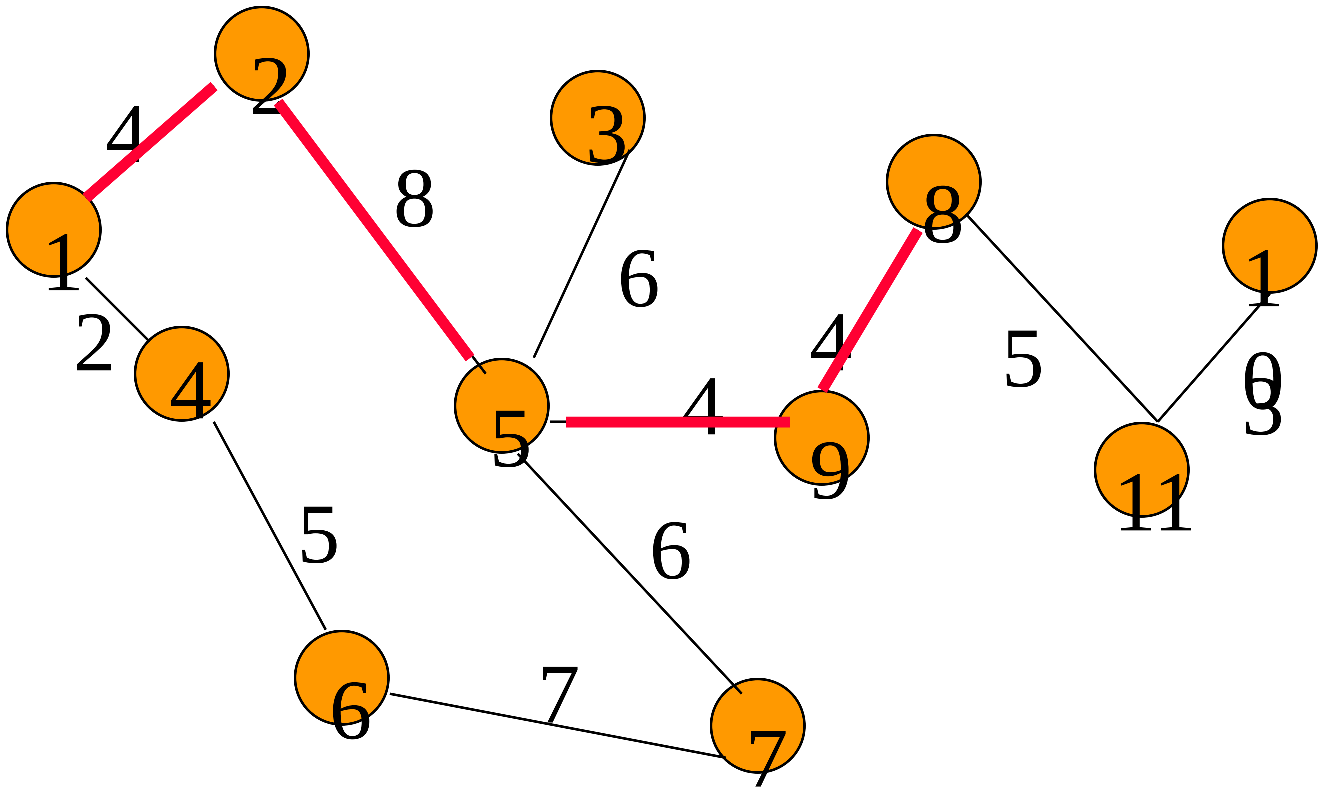
Graph Operations And Representation

Sample Graph Problems

- Path problems.
- Connectedness problems.
- Spanning tree problems.

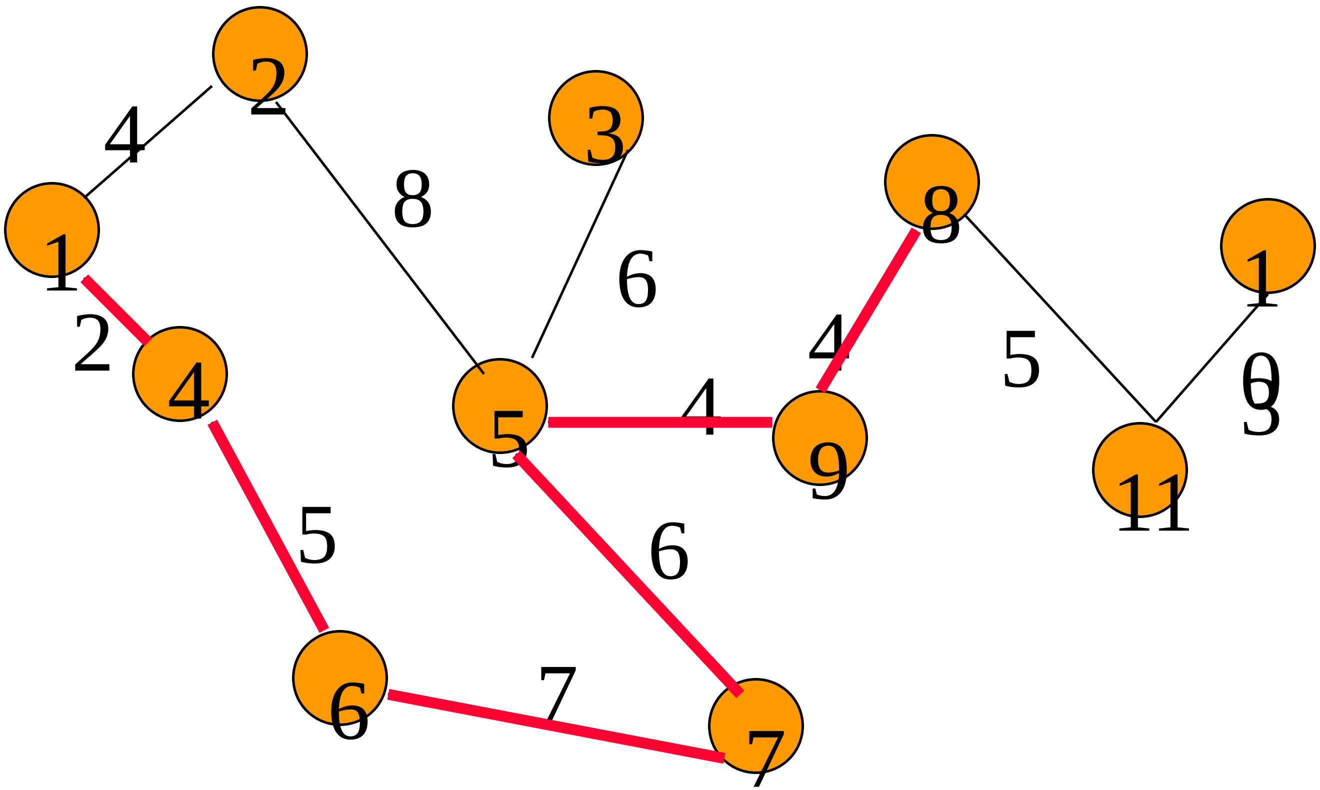
Path Finding

Path between 1 and 8.



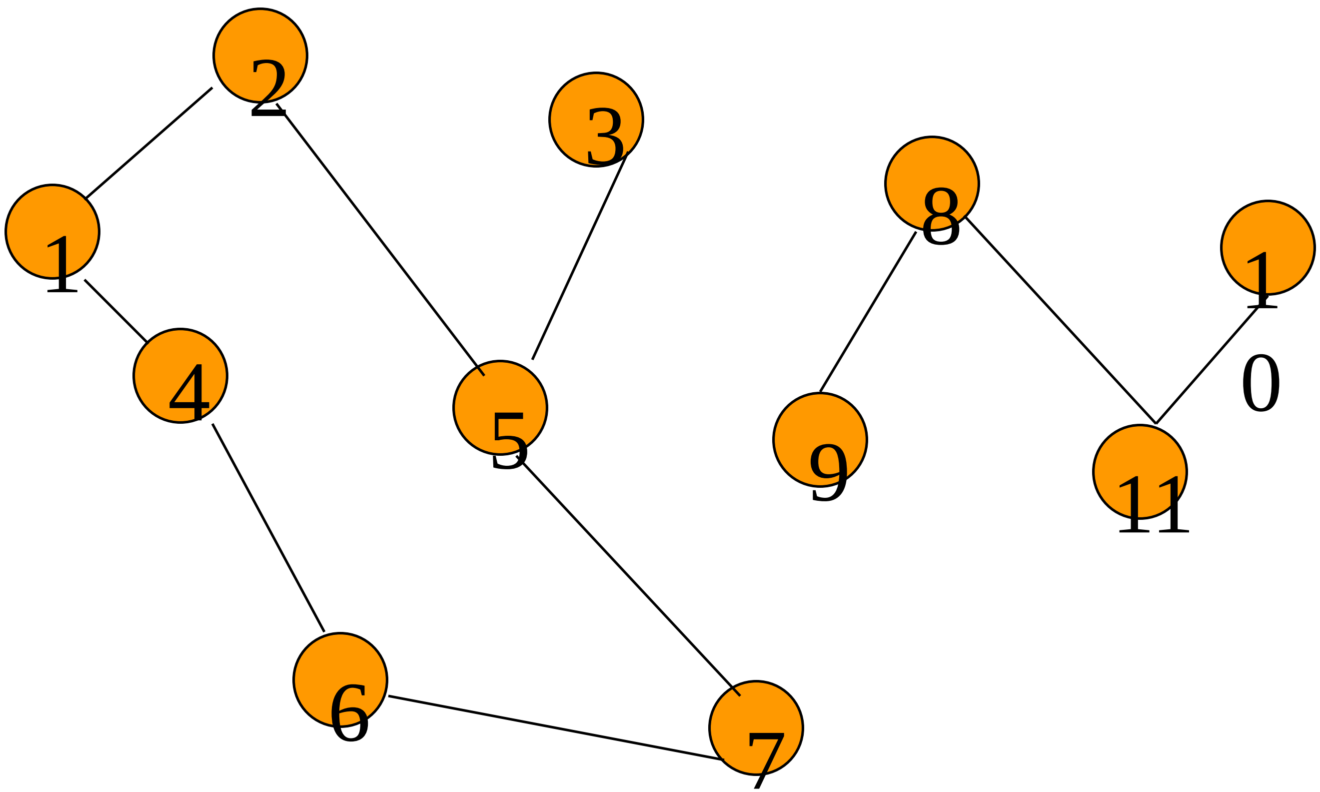
Path length is 20.

Another Path Between 1 and 8



Path length is 28.

Example Of No Path

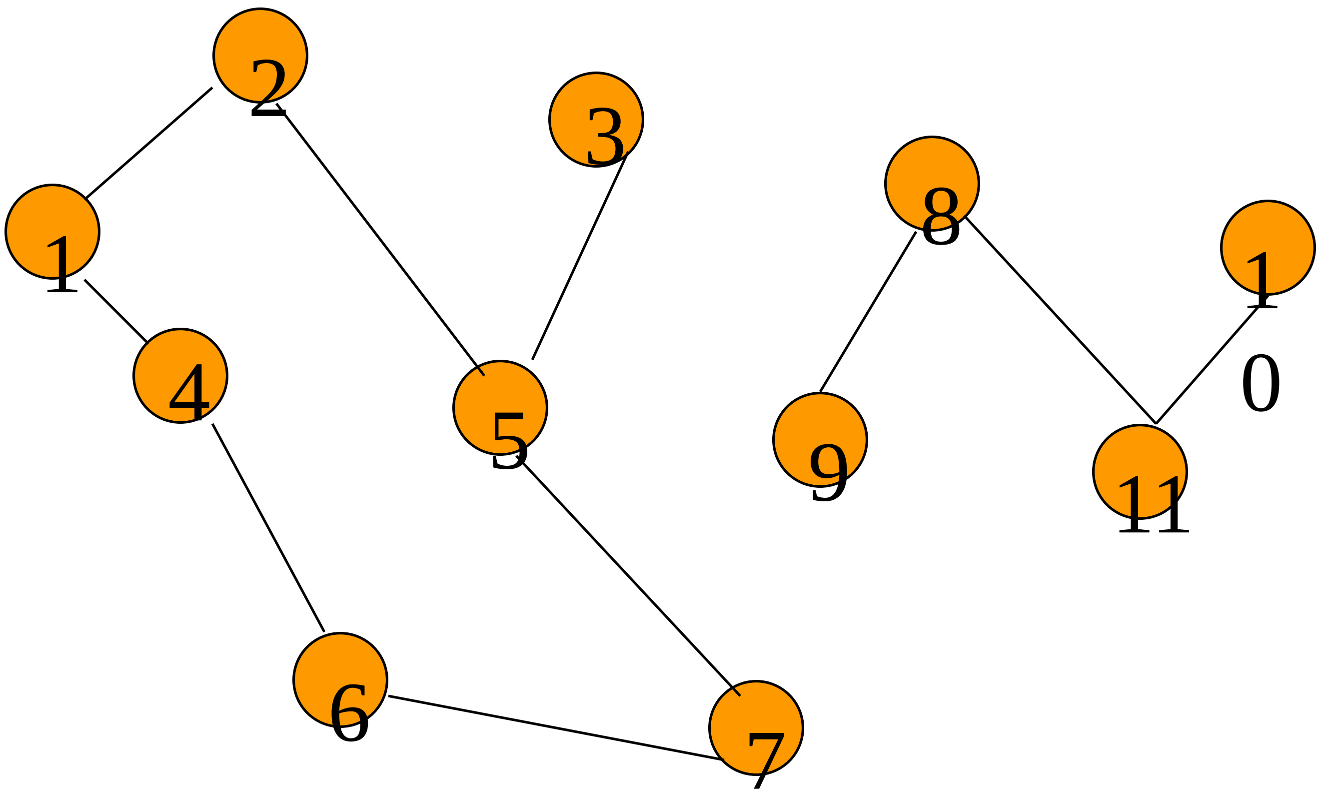


No path between 2 and 9.

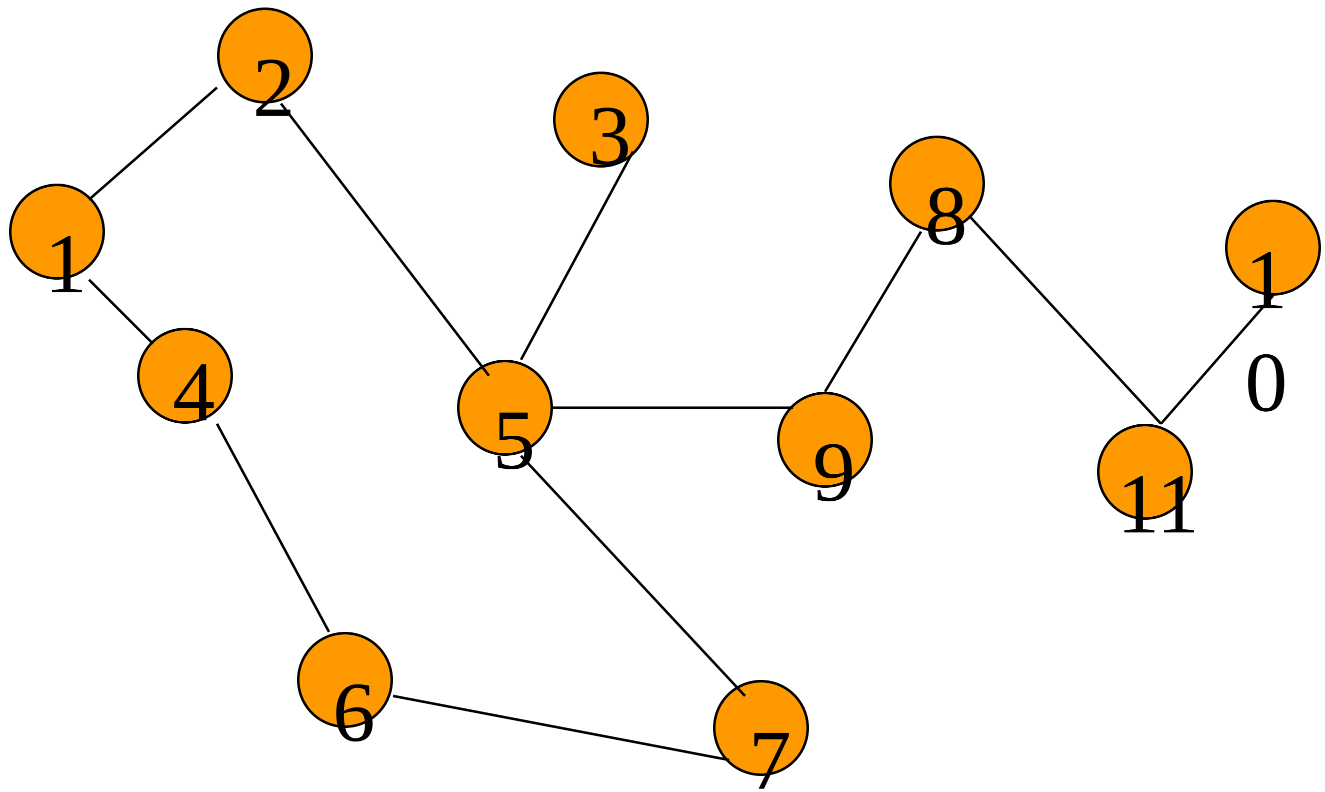
Connected Graph

- Undirected graph.
- There is a path between every pair of vertices.

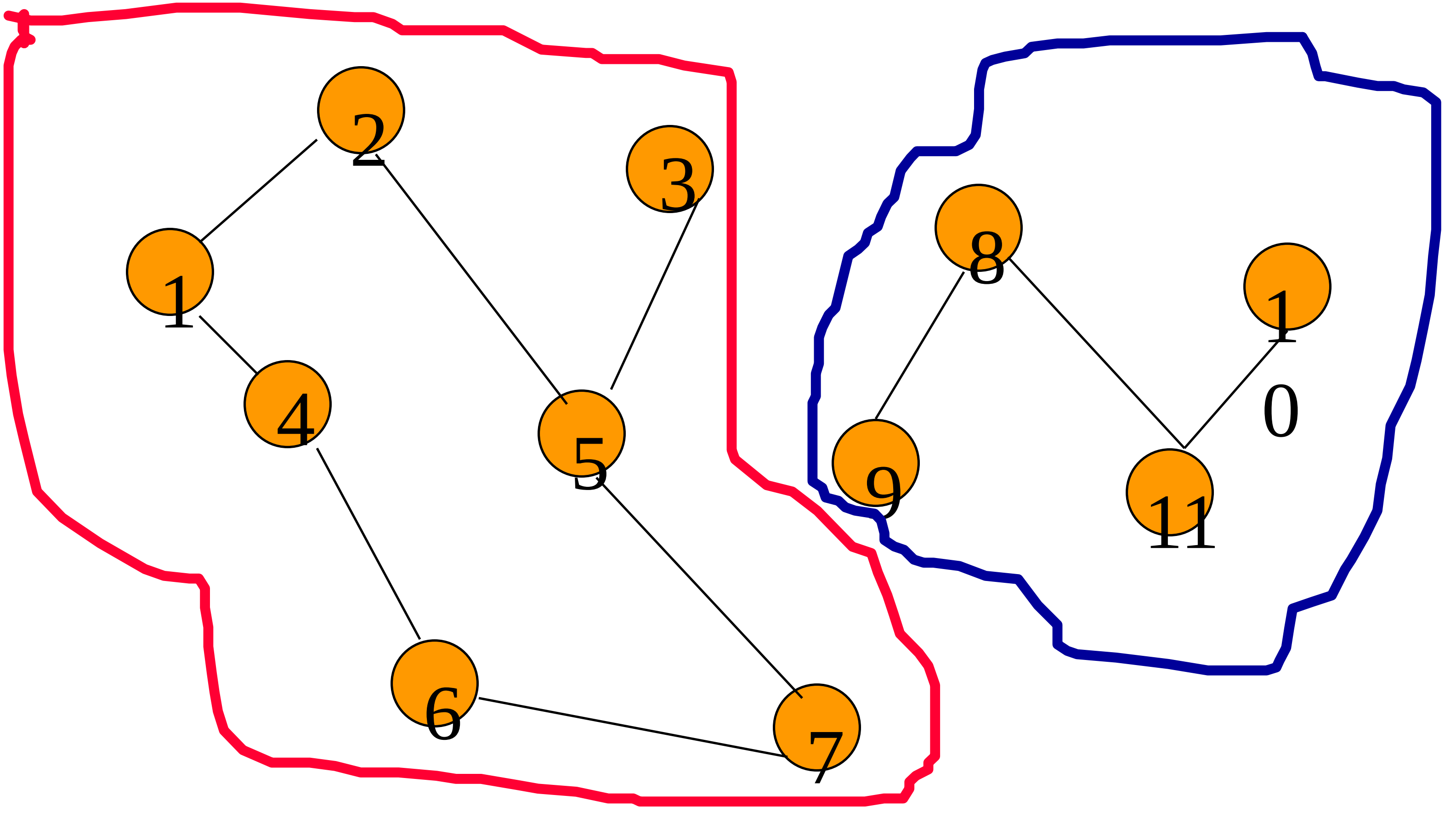
Example Of Not Connected



Connected Graph Example



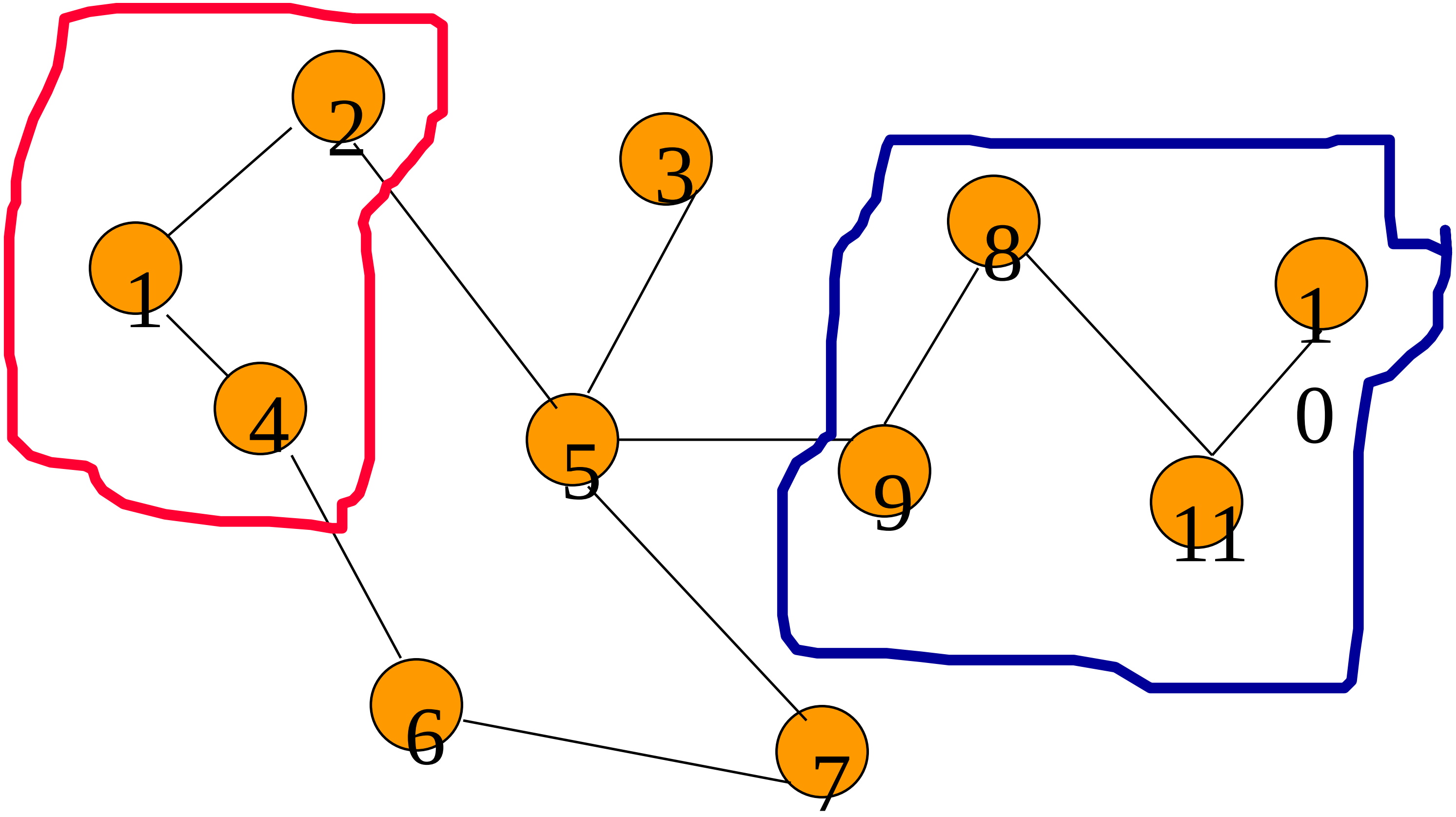
Connected Components



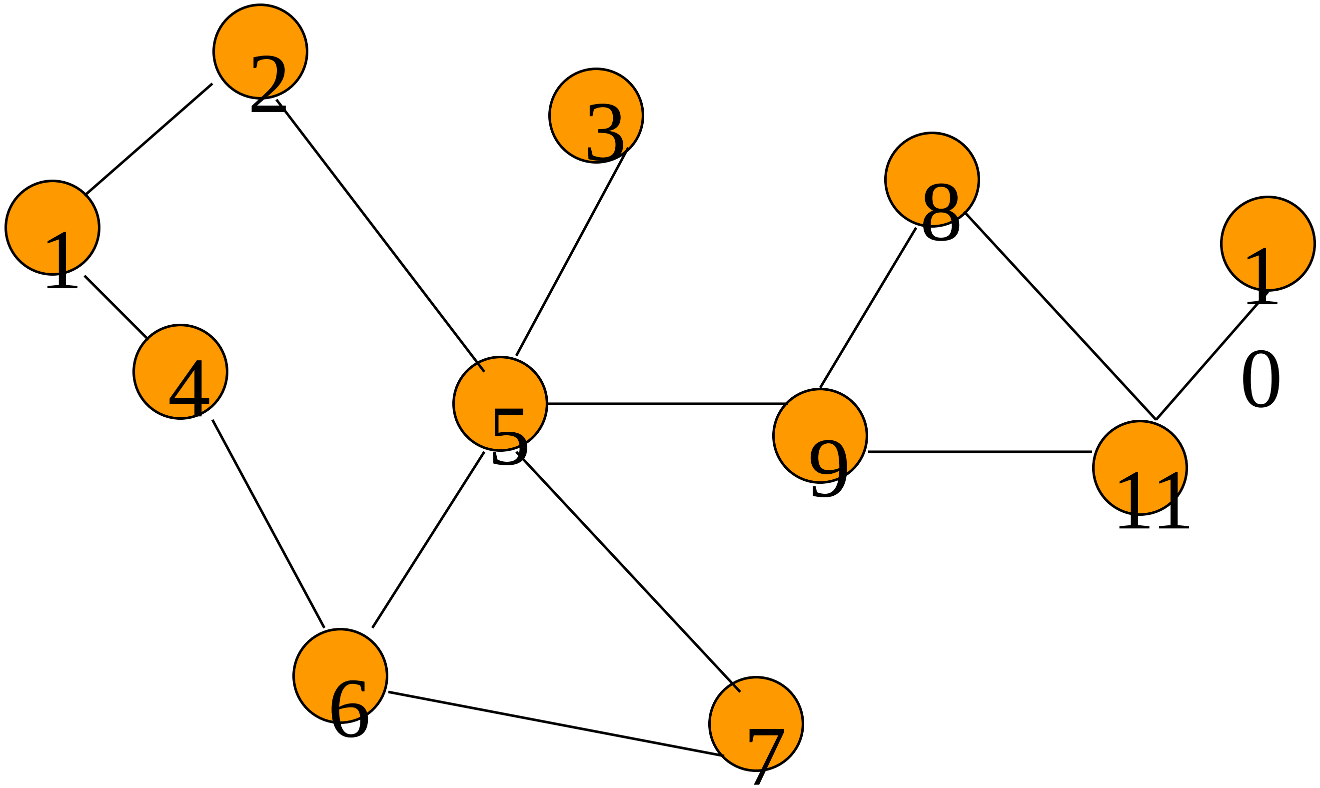
Connected Component

- A maximal subgraph that is connected.
  - Cannot add vertices and edges from original graph and retain connectedness.
- A connected graph has exactly 1 component.

Not A Component



Communication Network

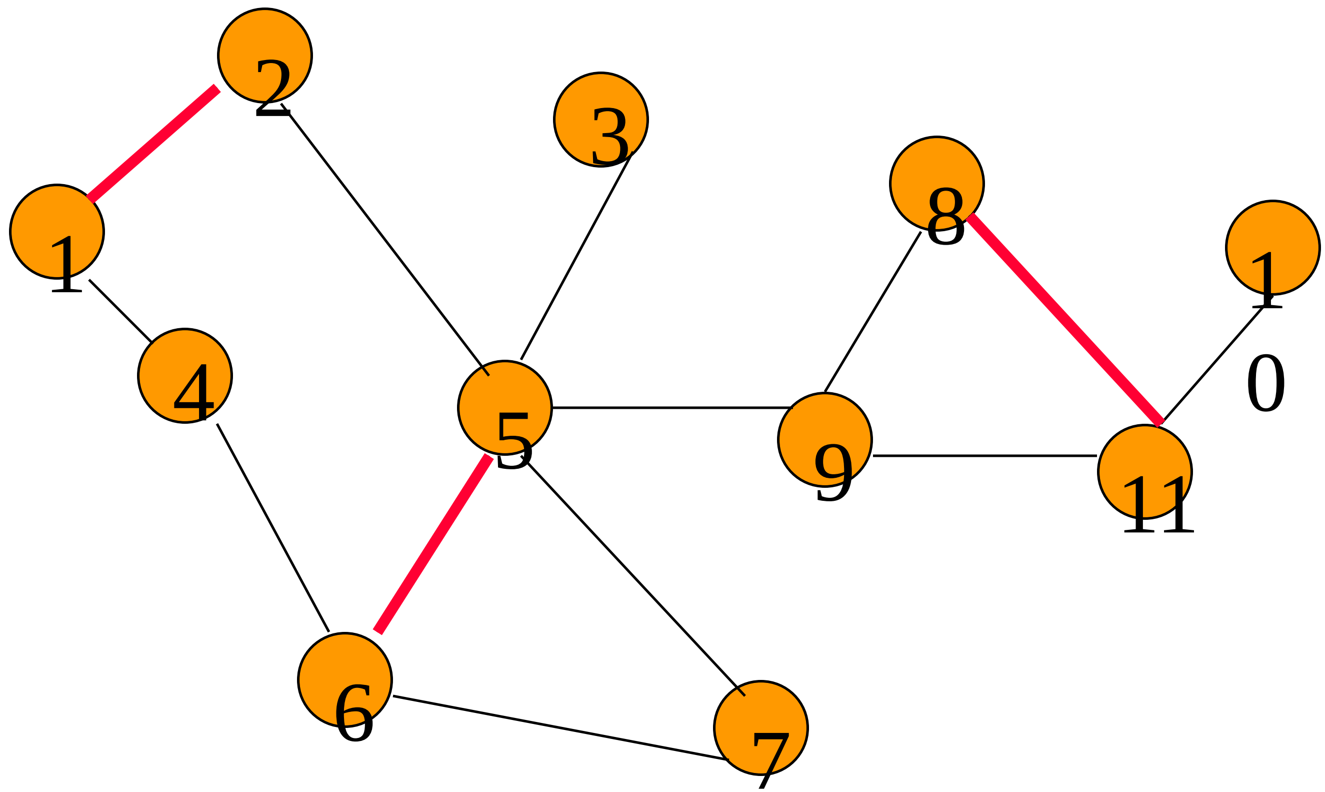


Each edge is a link that can be constructed (i.e., a feasible link).

Communication Network Problems

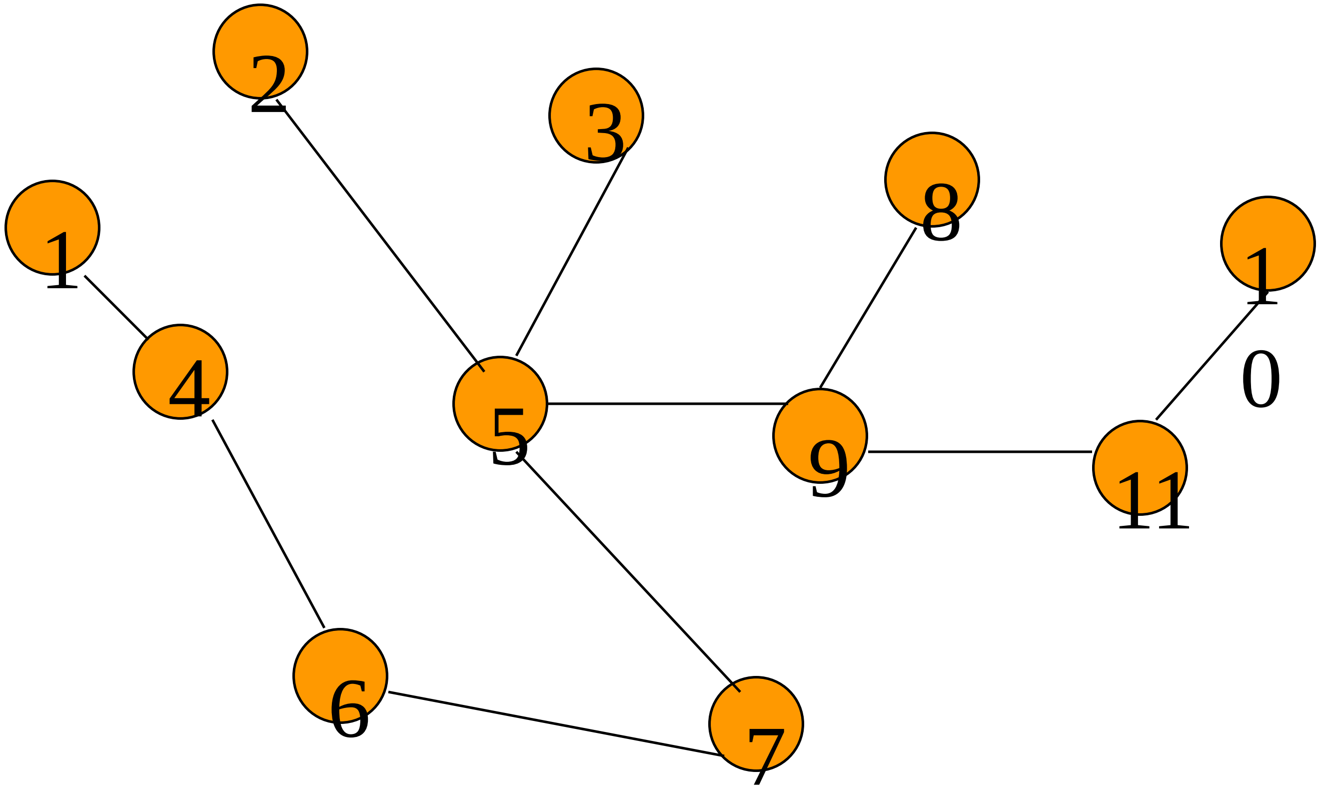
- Is the network connected?
  - Can we communicate between every pair of cities?
- Find the components.
- Want to construct smallest number of feasible links so that resulting network is connected.

Cycles And Connectedness



Removal of an edge that is on a cycle does not affect connectedness.

Cycles And Connectedness



Connected subgraph with all vertices and minimum number of edges has no cycles.

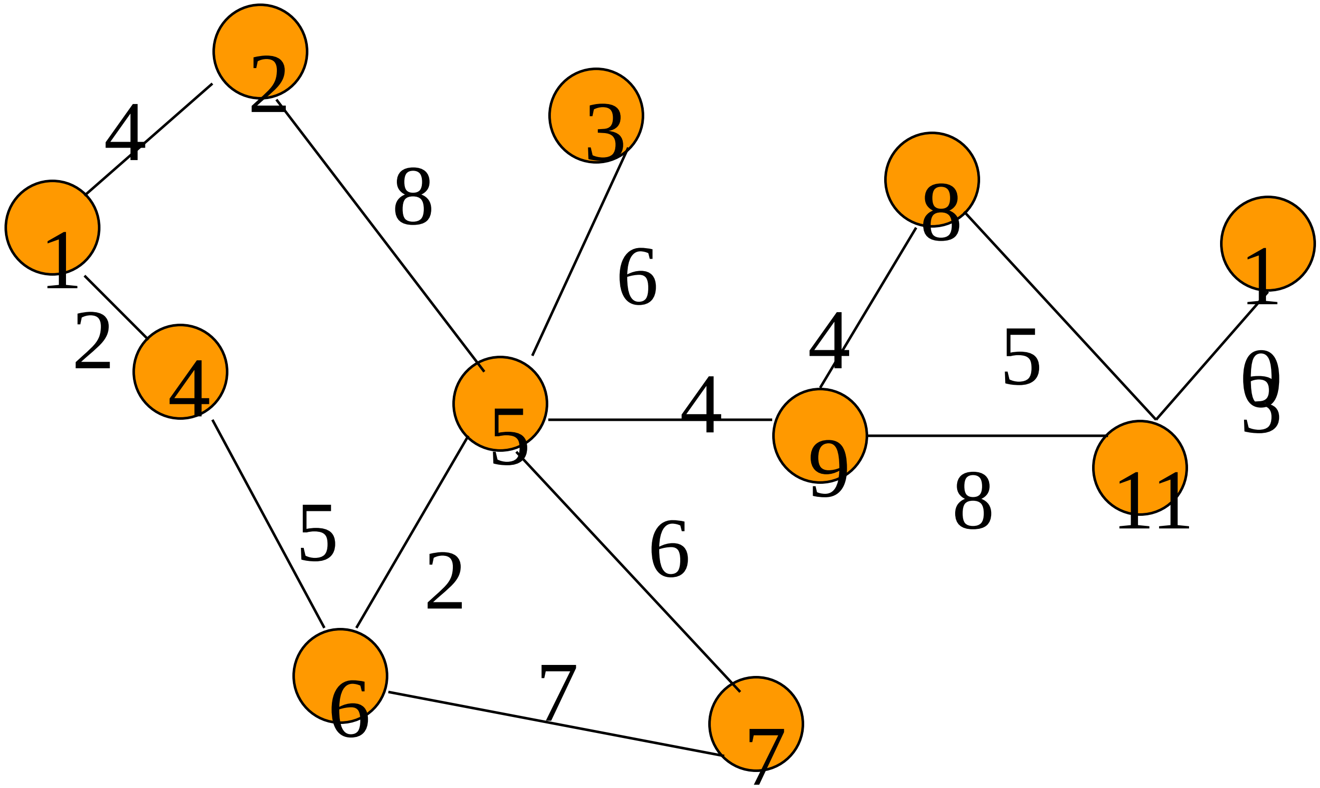
Tree

- Connected graph that has no cycles.
- n vertex connected graph with n-1 edges.

Spanning Tree

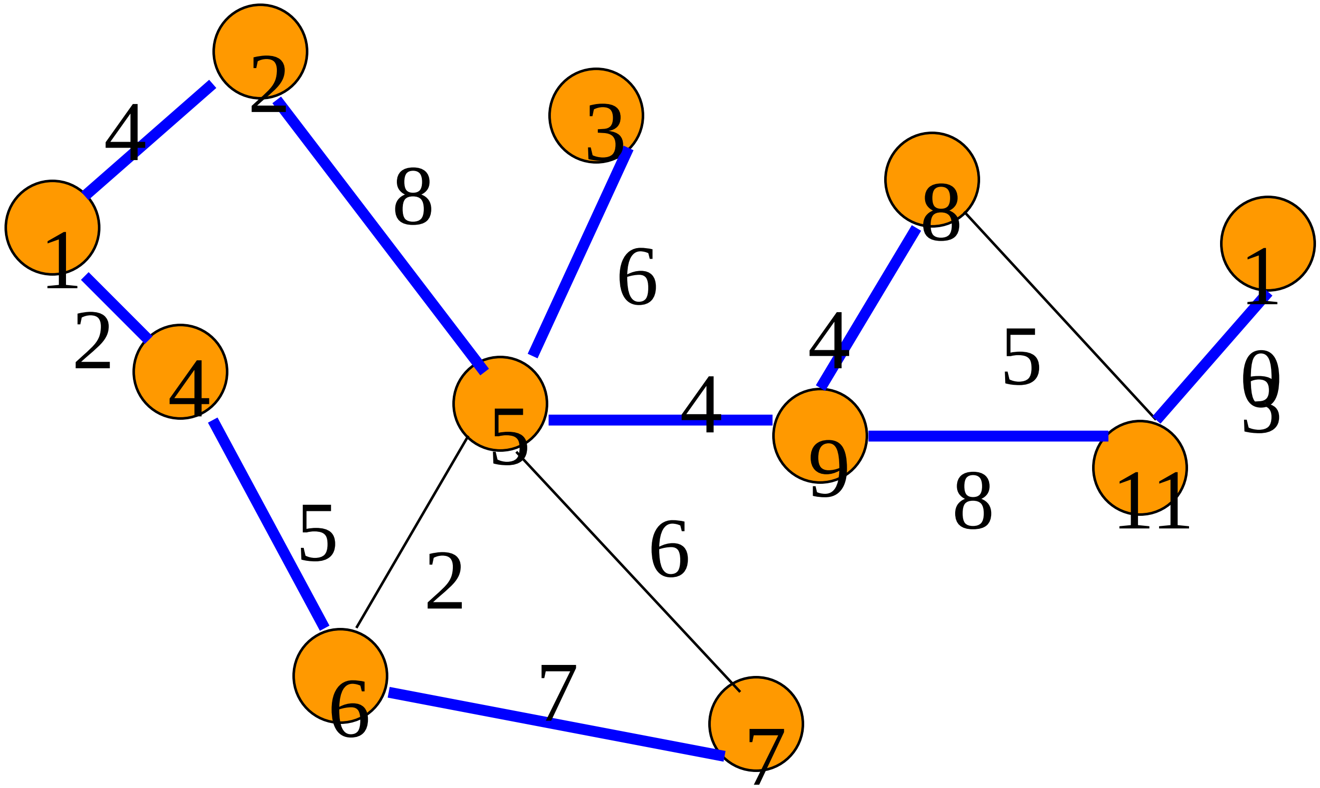
- Subgraph that includes all vertices of the original graph.
- Subgraph is a tree.
  - If original graph has n vertices, the spanning tree has n vertices and n-1 edges.

Minimum Cost Spanning Tree



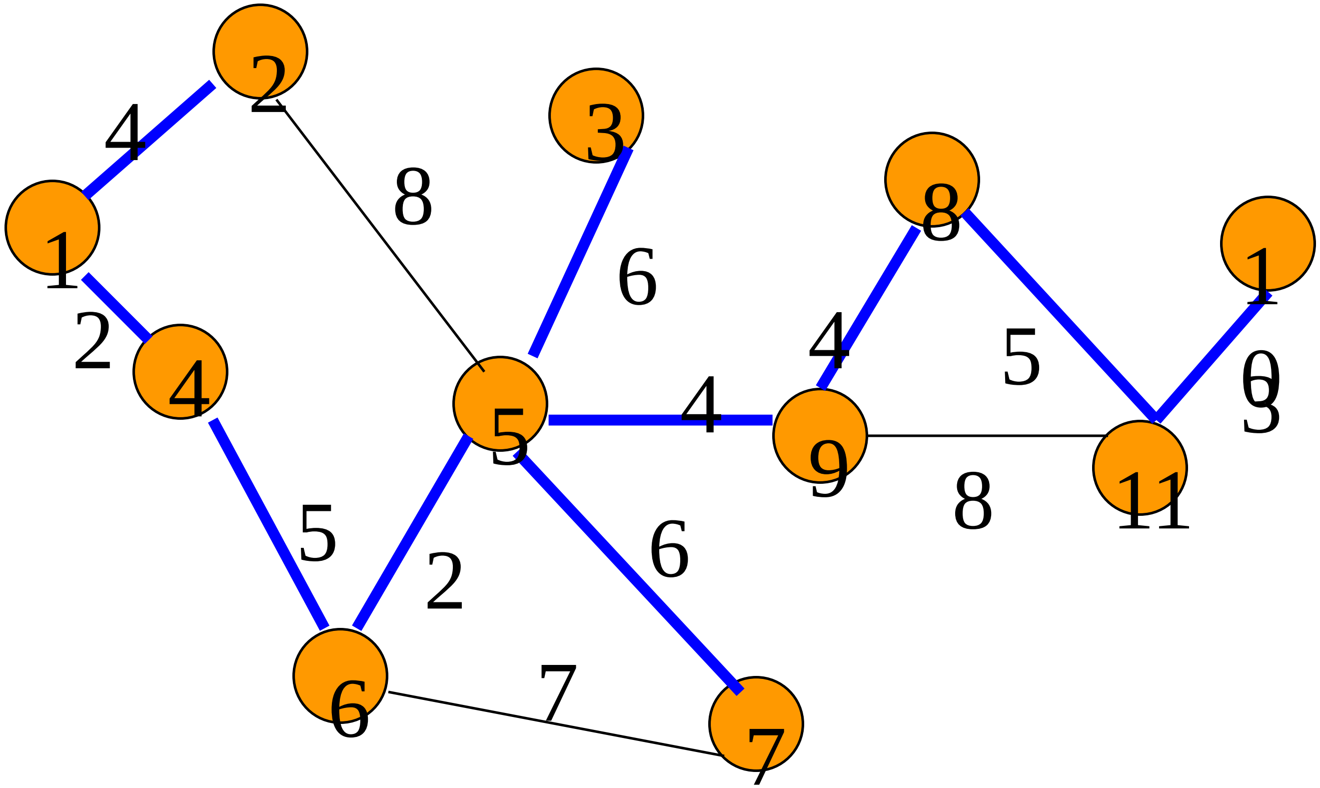
- Tree cost is sum of edge weights/costs.

A Spanning Tree



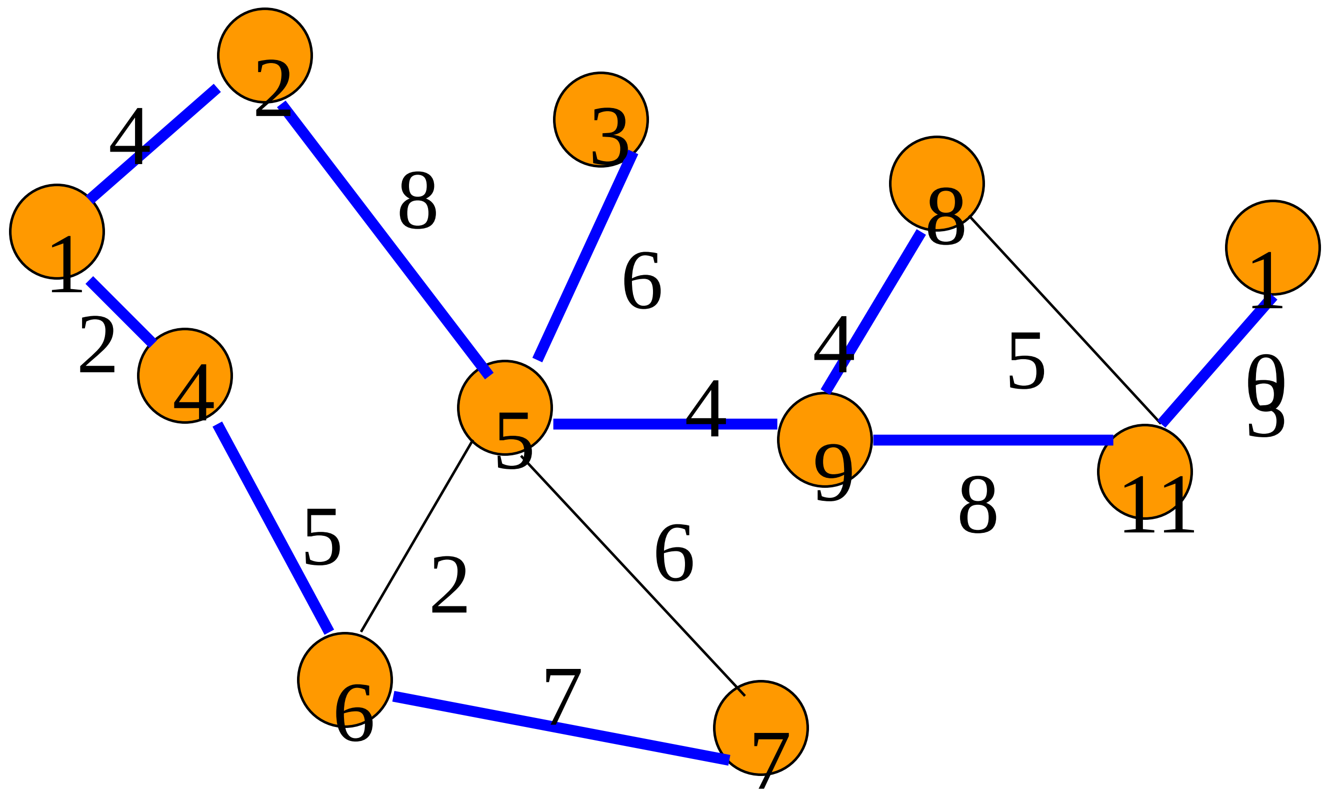
Spanning tree cost = 51.

Minimum Cost Spanning Tree



Spanning tree cost = 41.

A Wireless Broadcast Tree



Source = 1, weights = needed power.

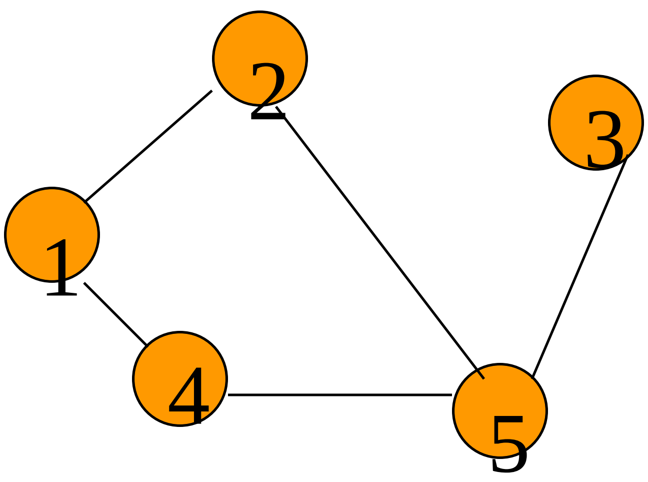
Cost = 4 + 8 + 5 + 6 + 7 + 8 + 3 = 41.

Graph Representation

- Adjacency Matrix
- Adjacency Lists
  - Linked Adjacency Lists
  - Array Adjacency Lists

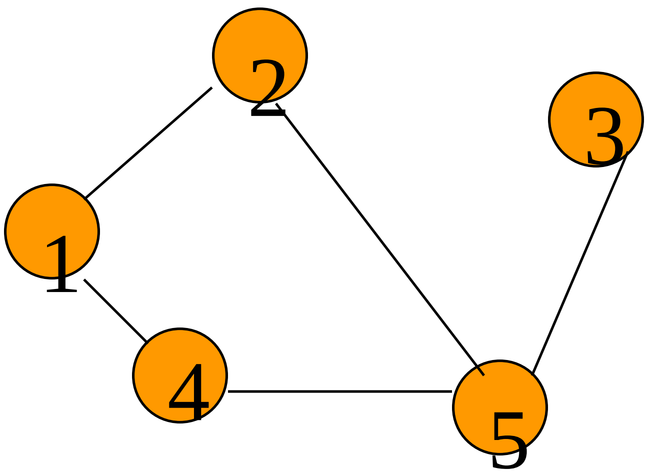
Adjacency Matrix

- 0/1 n x n matrix, where n = # of vertices
- A(i,j) = 1 iff (i,j) is an edge



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

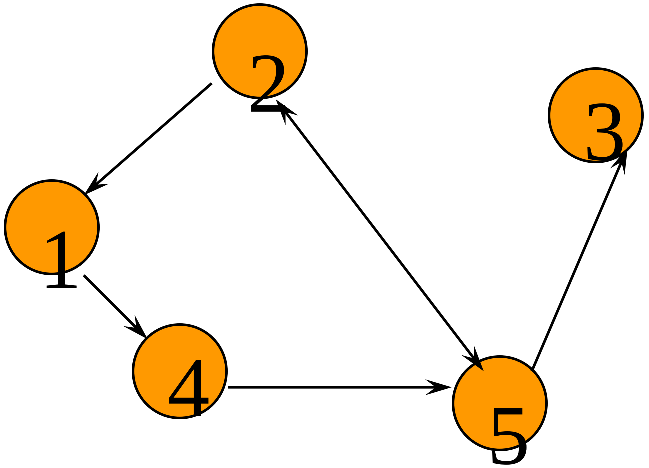
Adjacency Matrix Properties



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

- Diagonal entries are zero.
- Adjacency matrix of an undirected graph is symmetric.
    - A(i,j) = A(j,i) for all i and j.

Adjacency Matrix (Digraph)



	1	2	3	4	5
1	0	0	0	1	0
2	1	0	0	0	1
3	0	0	0	0	0
4	0	0	0	0	1
5	0	1	1	0	0

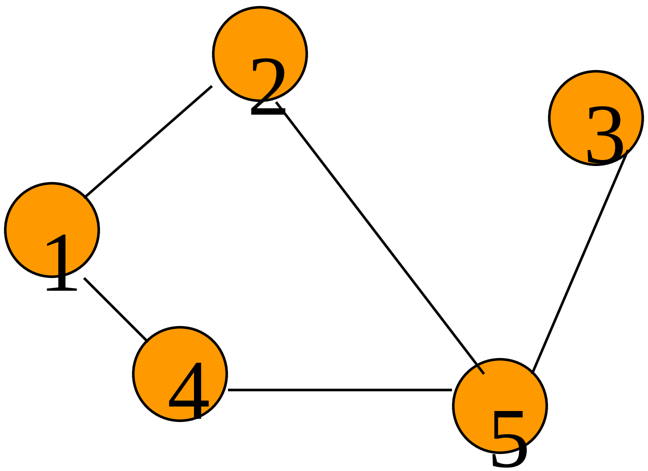
- Diagonal entries are zero.
- Adjacency matrix of a digraph need not be symmetric.

Adjacency Matrix

- n^2 bits of space
- For an undirected graph, may store only lower or upper triangle (exclude diagonal).
  - $(n-1)n/2$ bits
- $O(n)$ time to find vertex degree and/or vertices adjacent to a given vertex.

Adjacency Lists

- Adjacency list for vertex i is a linear list of vertices adjacent from vertex i.
- An array of n adjacency lists.



aList[1] = (2,4)

aList[2] = (1,5)

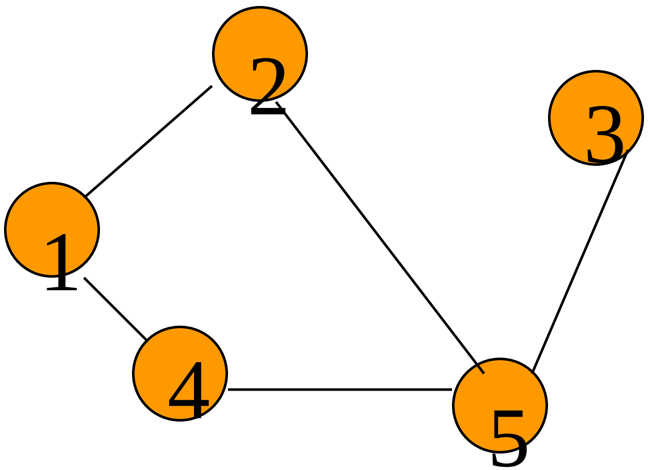
aList[3] = (5)

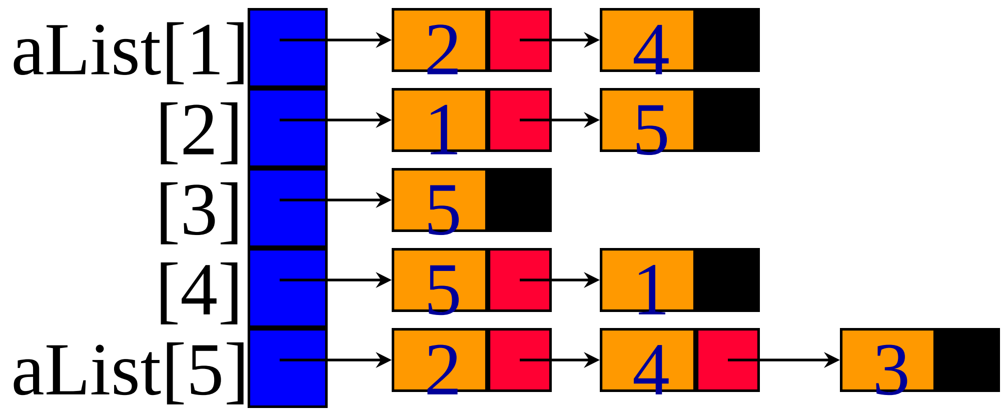
aList[4] = (5,1)

aList[5] = (2,4,3)

Linked Adjacency Lists

- Each adjacency list is a chain.





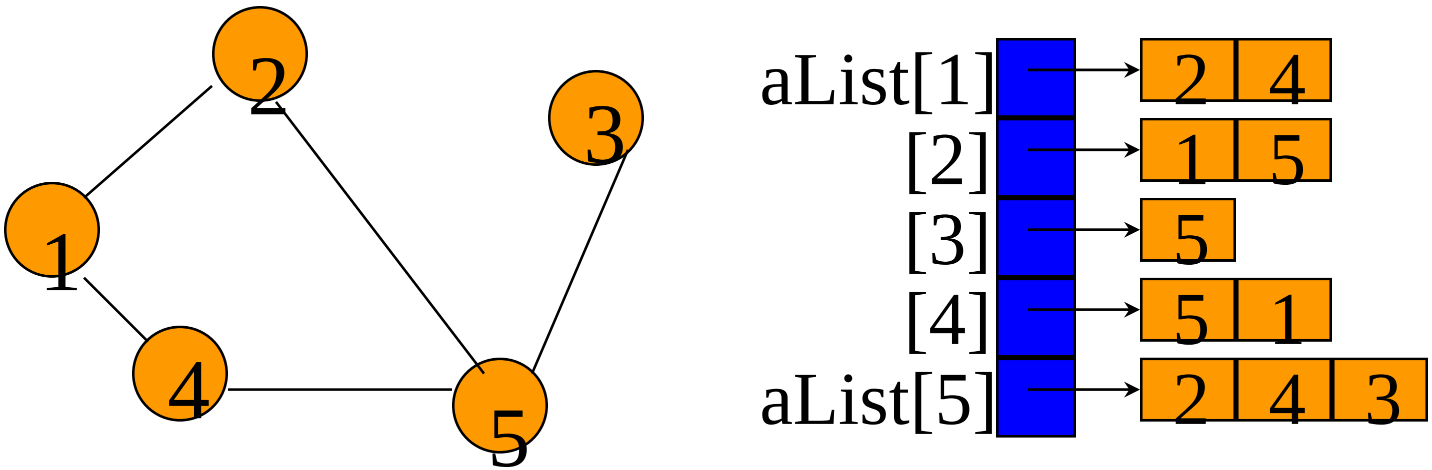
Array Length = n

# of chain nodes = 2e (undirected graph)

# of chain nodes = e (digraph)

Array Adjacency Lists

- Each adjacency list is an array list.



Array Length = n

# of list elements = 2e (undirected graph)

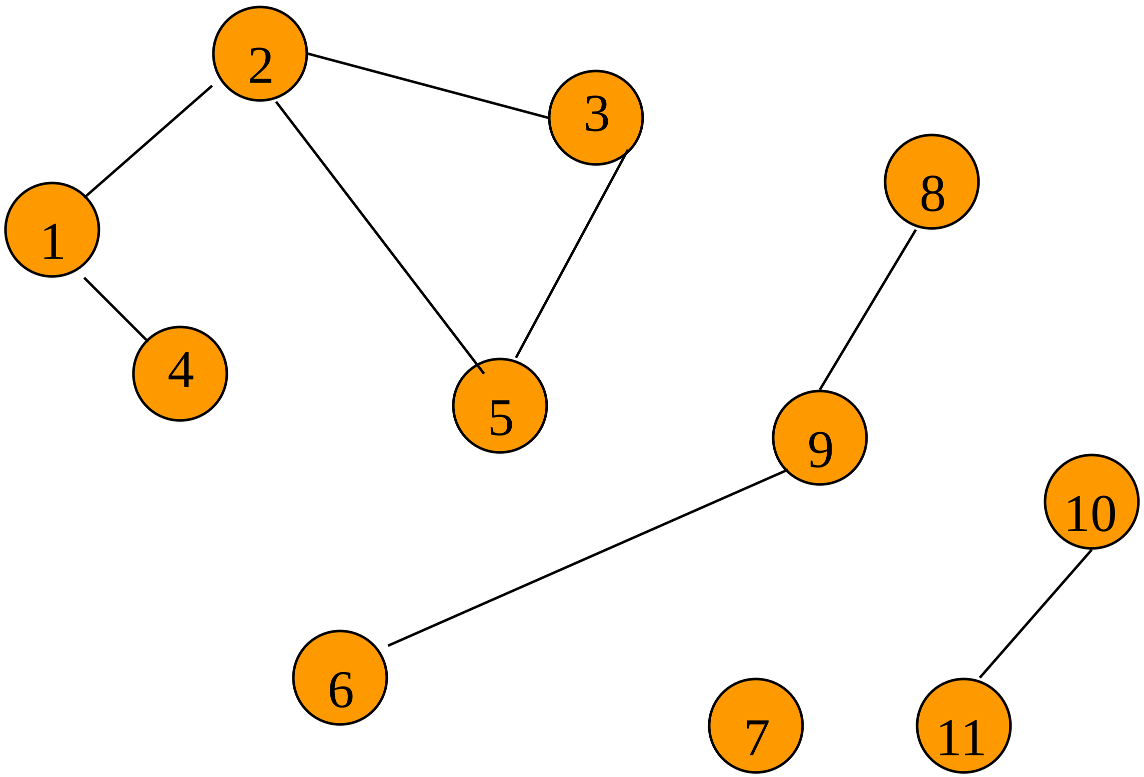
# of list elements = e (digraph)

Weighted Graphs

- Cost adjacency matrix.
  - C(i,j) = cost of edge (i,j)
- Adjacency lists => each list element is a pair (adjacent vertex, edge weight)

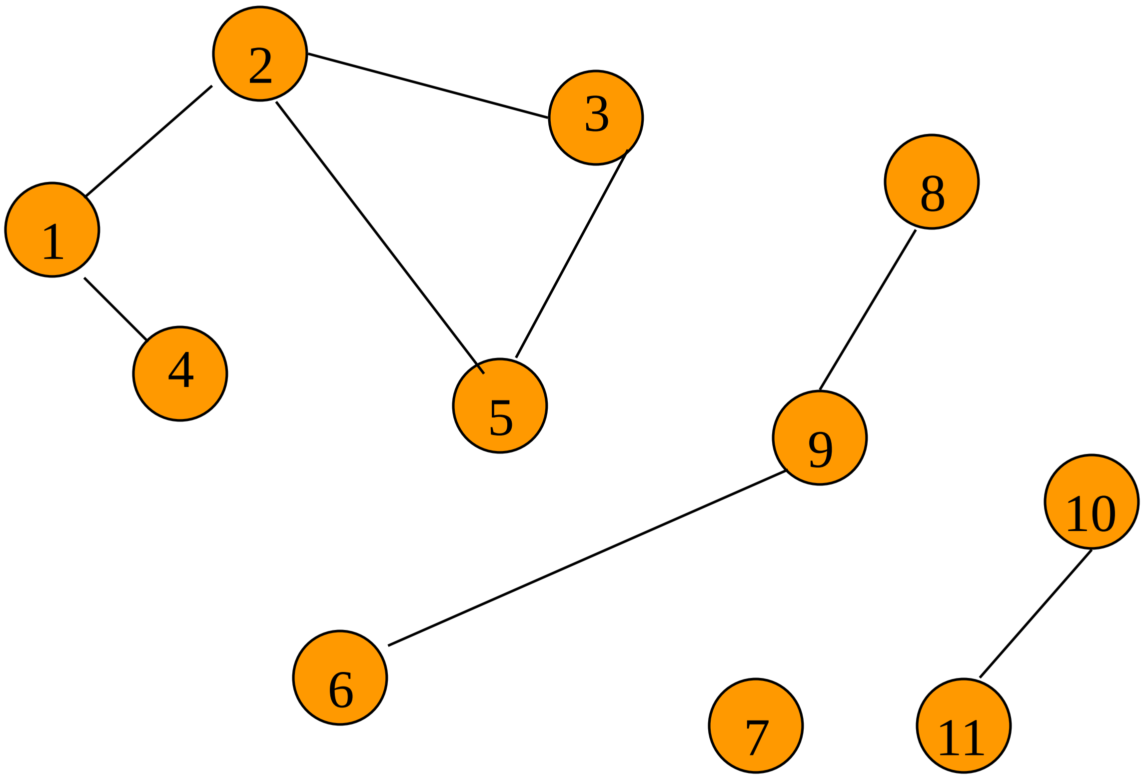
Graph Search Methods

- A vertex u is reachable from vertex v iff there is a path from v to u.



Graph Search Methods

- A search method starts at a given vertex v and visits/labels/marks every vertex that is reachable from v.



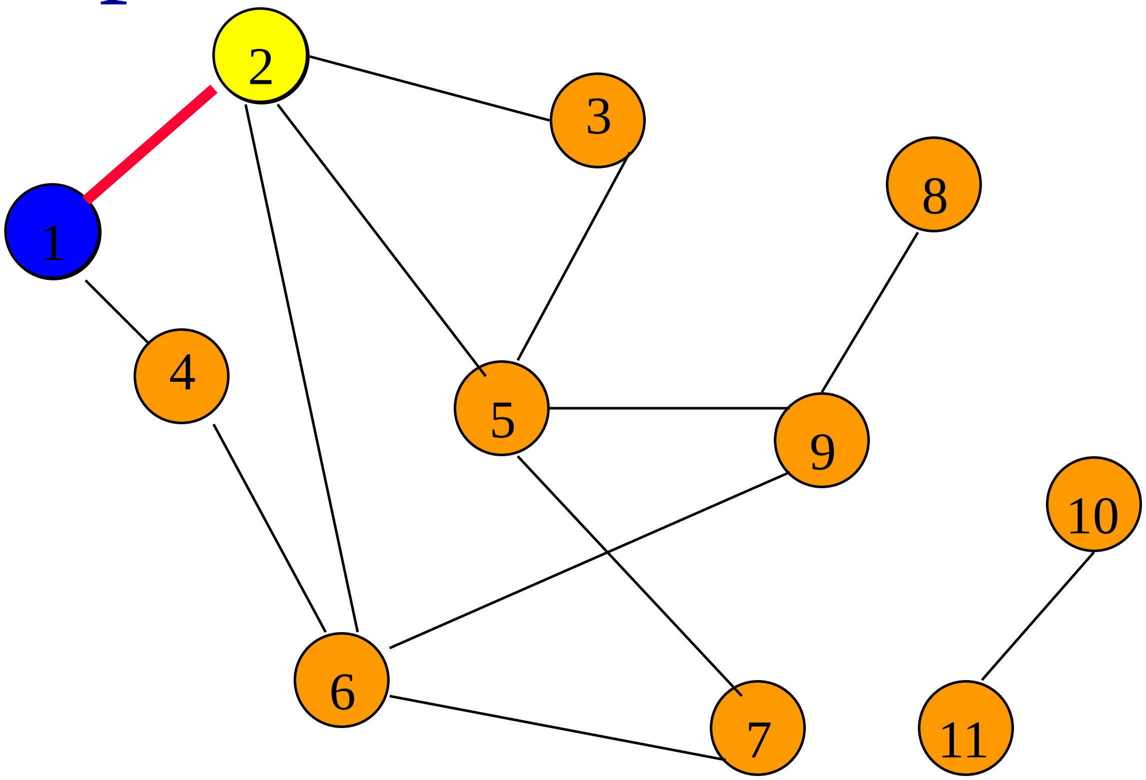
Graph Search Methods

- Many graph problems solved using a search method.
  - Path from one vertex to another.
  - Is the graph connected?
  - Find a spanning tree.
  - Etc.
- Commonly used search methods:
  - Depth-first search.
  - Breadth-first search.

Depth-First Search

```
dfs(v)
{
  Label vertex v as reached.
  for (each unreached vertex u
                        adjacenct from v)
    dfs(u);
}
```

Depth-First Search Example

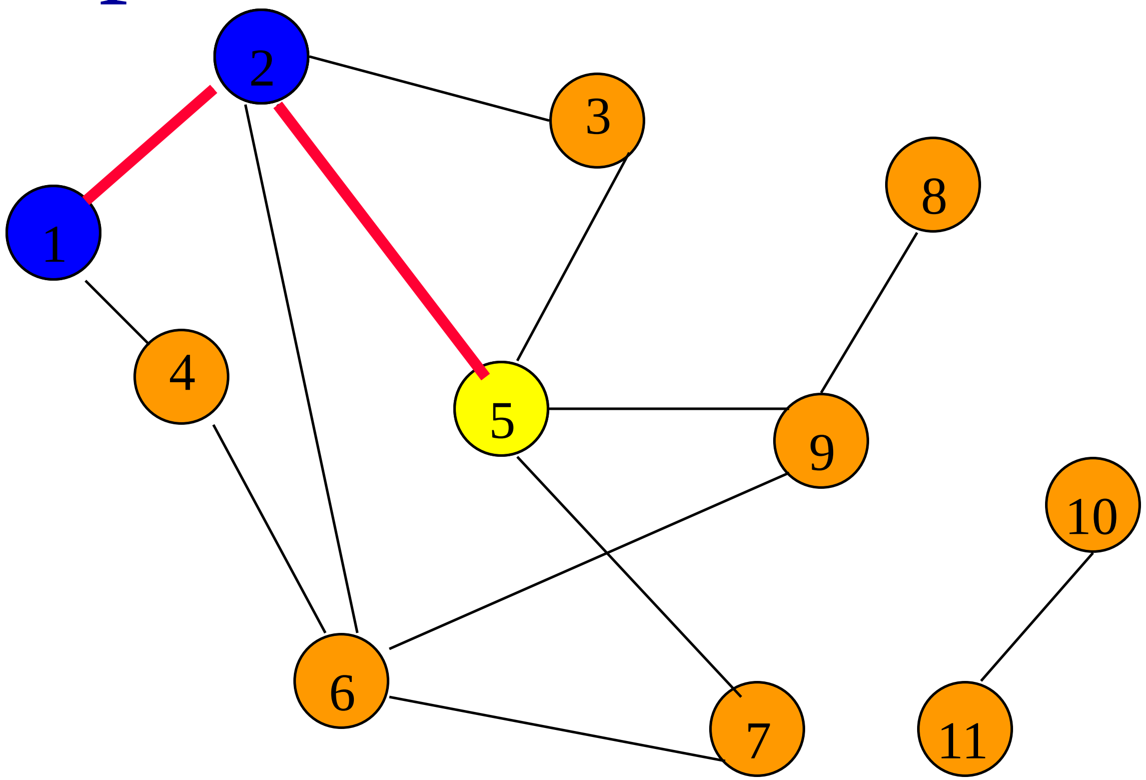


Start search at vertex 1.

Label vertex 1 and do a depth first search from either 2 or 4.

Suppose that vertex 2 is selected.

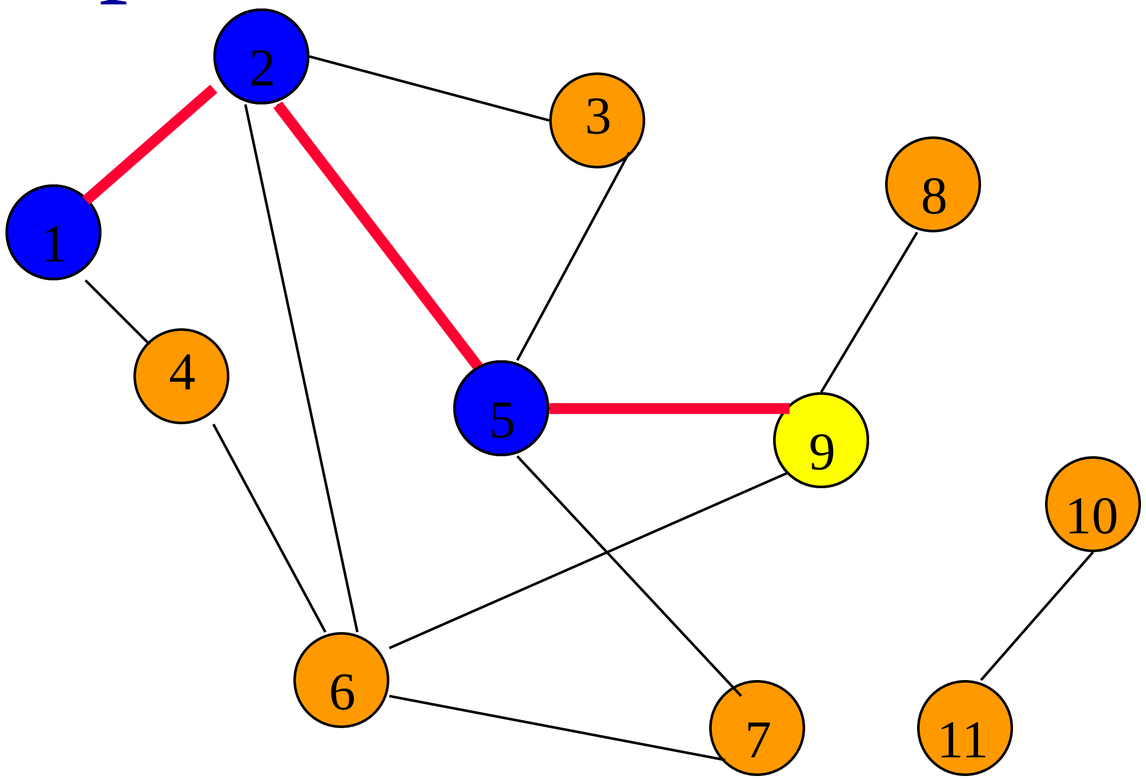
Depth-First Search Example



Label vertex 2 and do a depth first search from either 3, 5, or 6.

Suppose that vertex 5 is selected.

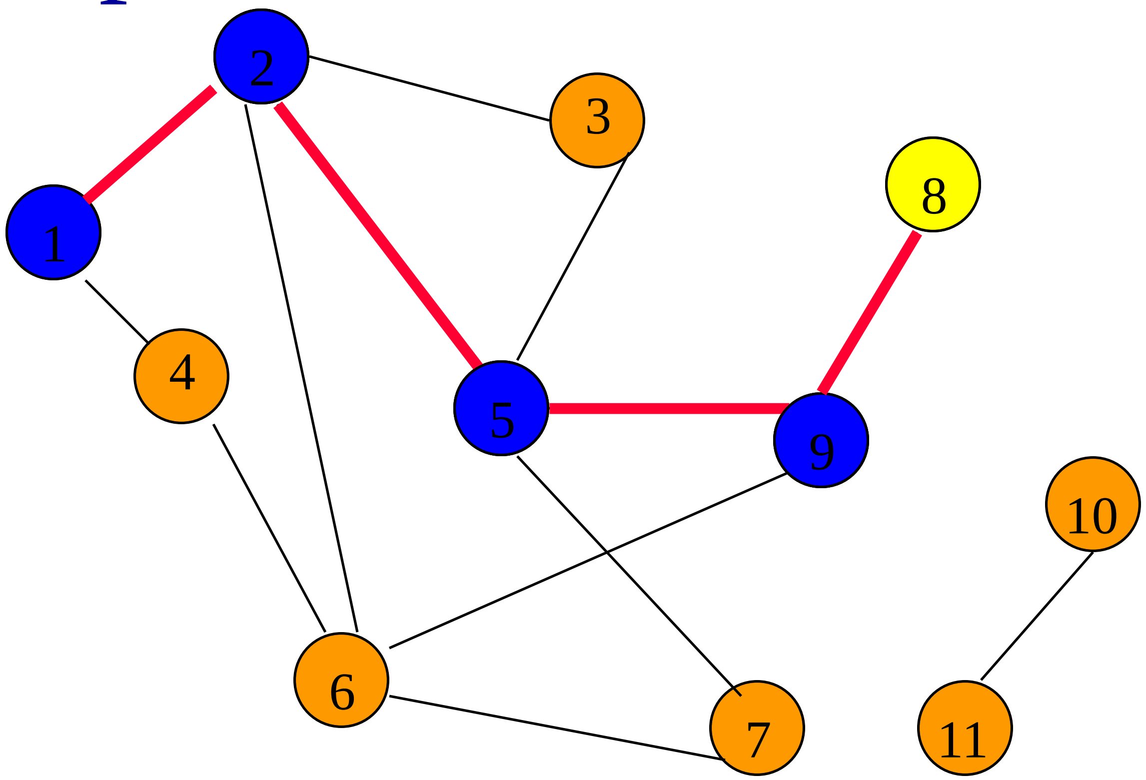
Depth-First Search Example



Label vertex 5 and do a depth first search from either 3, 7, or 9.

Suppose that vertex 9 is selected.

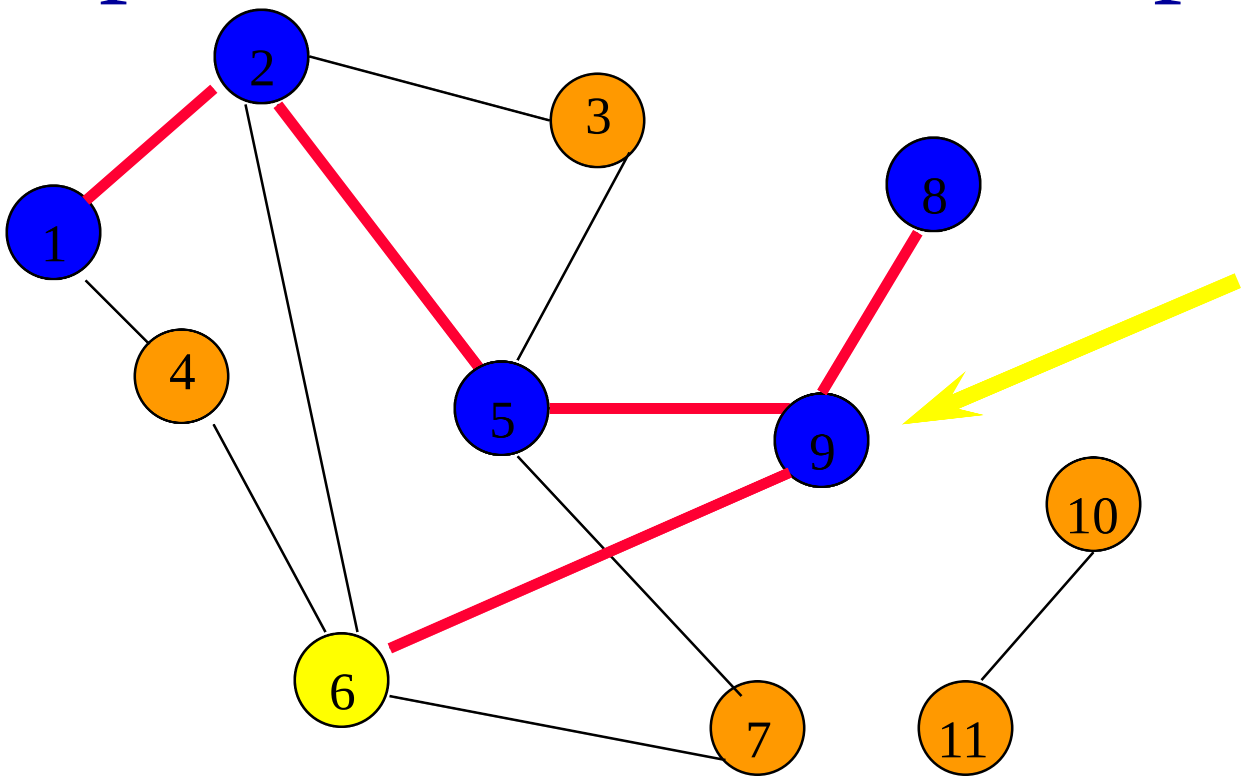
Depth-First Search Example



Label vertex 9 and do a depth first search from either 6 or 8.

Suppose that vertex 8 is selected.

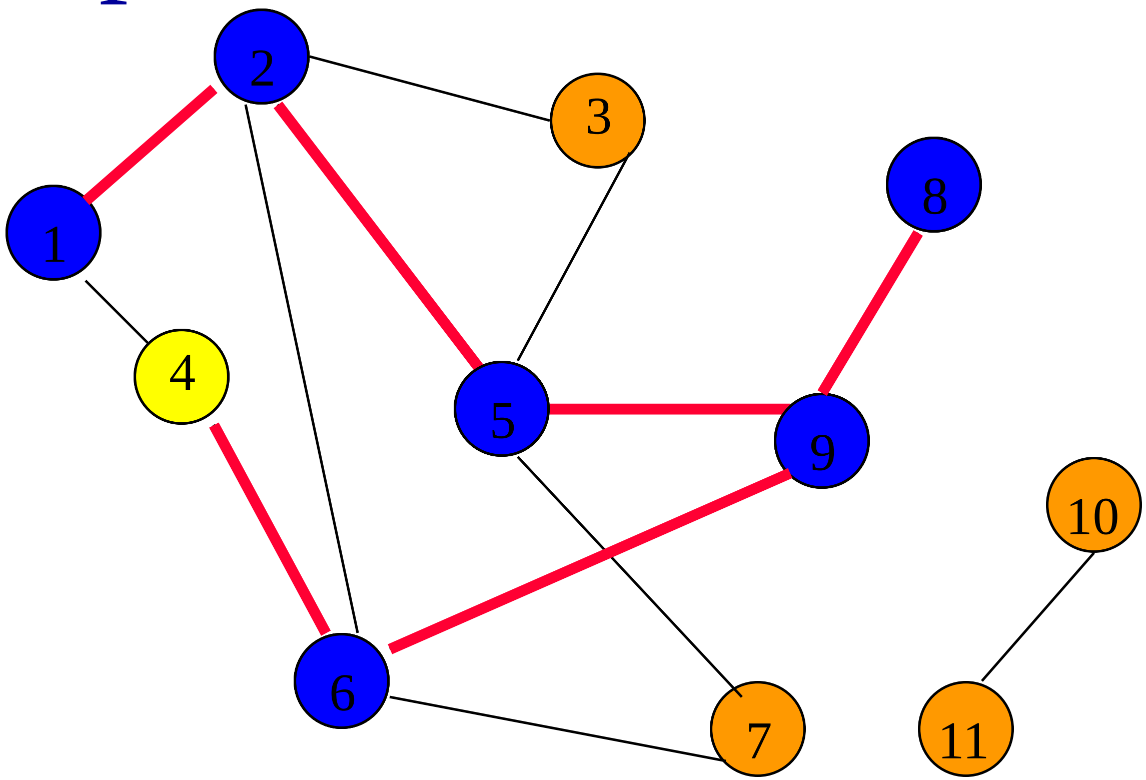
Depth-First Search Example



Label vertex 8 and return to vertex 9.

From vertex 9 do a DFS(6).

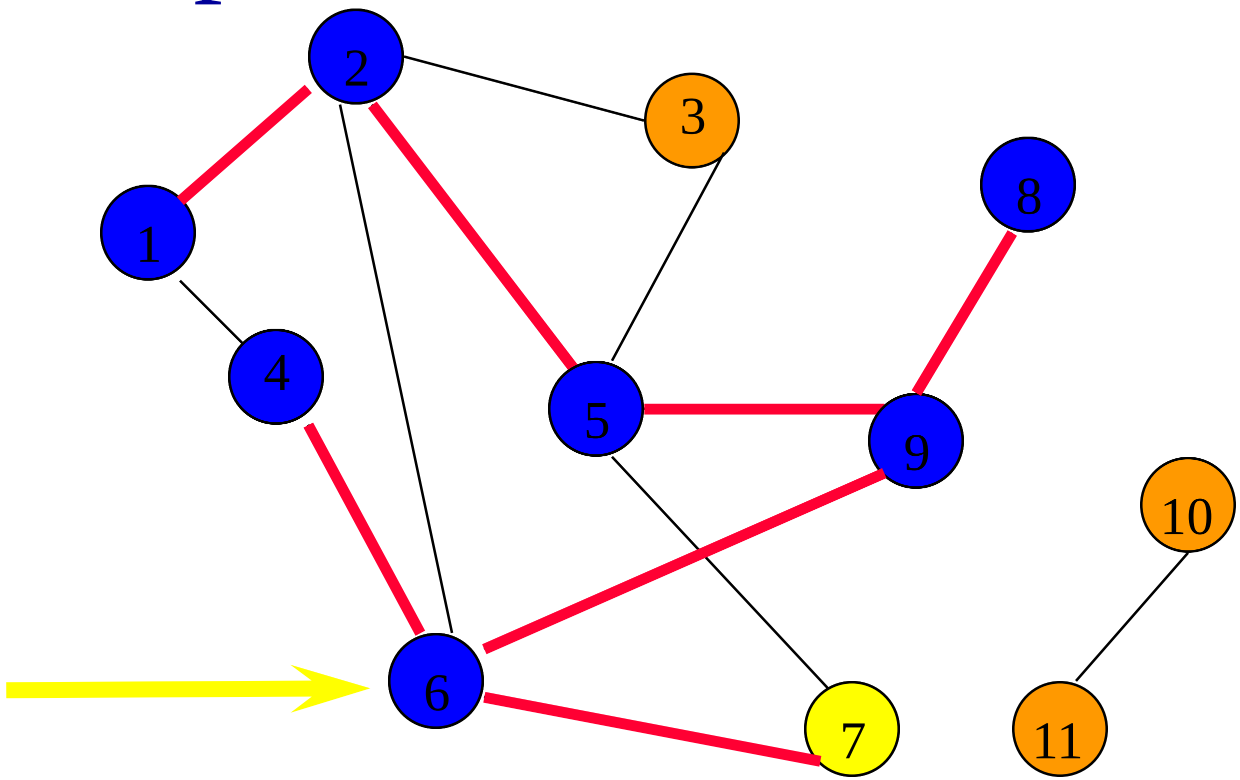
Depth-First Search Example



Label vertex 6 and do a depth first search from either 4 or 7.

Suppose that vertex 4 is selected.

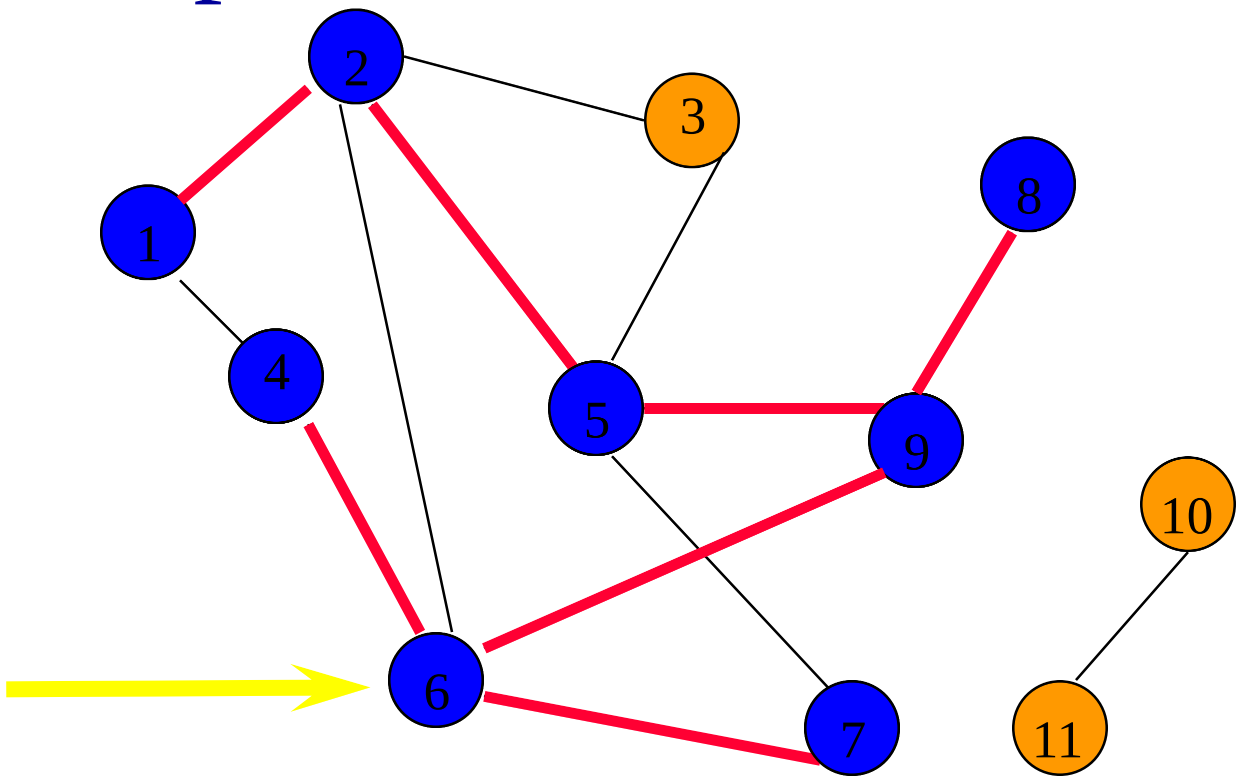
Depth-First Search Example



Label vertex 4 and return to 6.

From vertex 6 do a dfs(7).

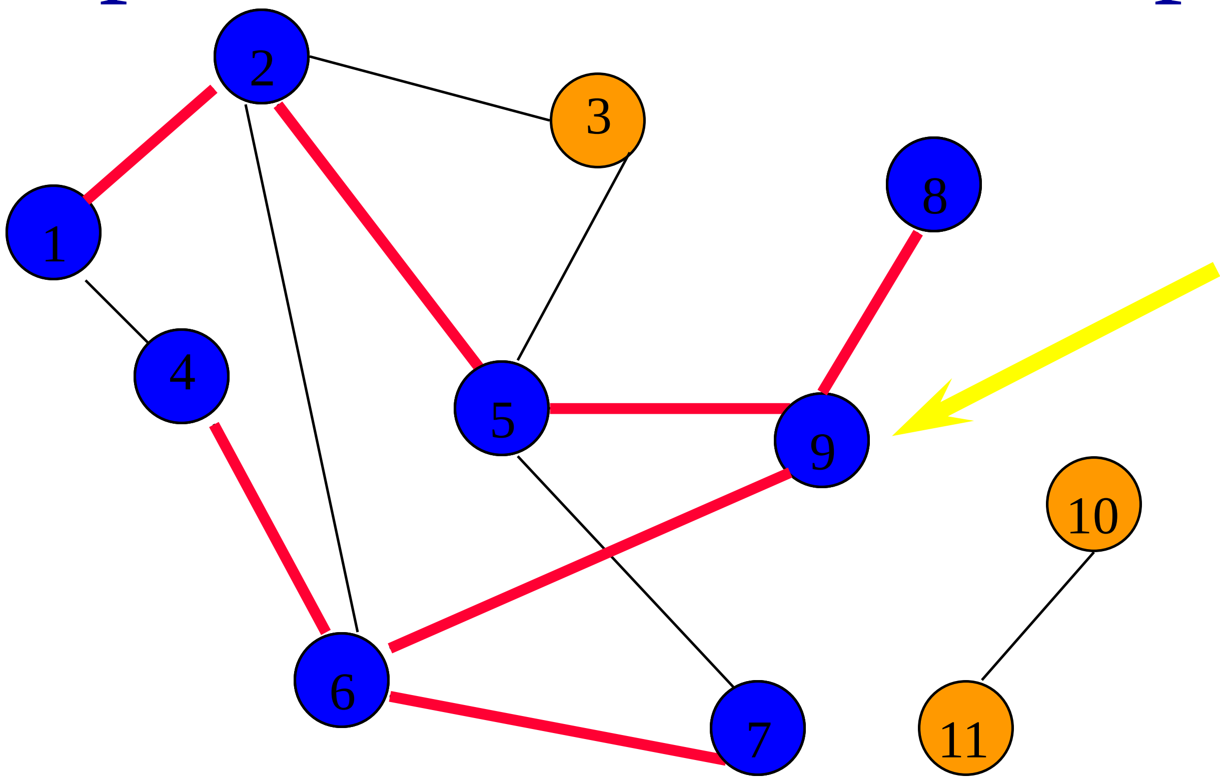
Depth-First Search Example



Label vertex 7 and return to 6.

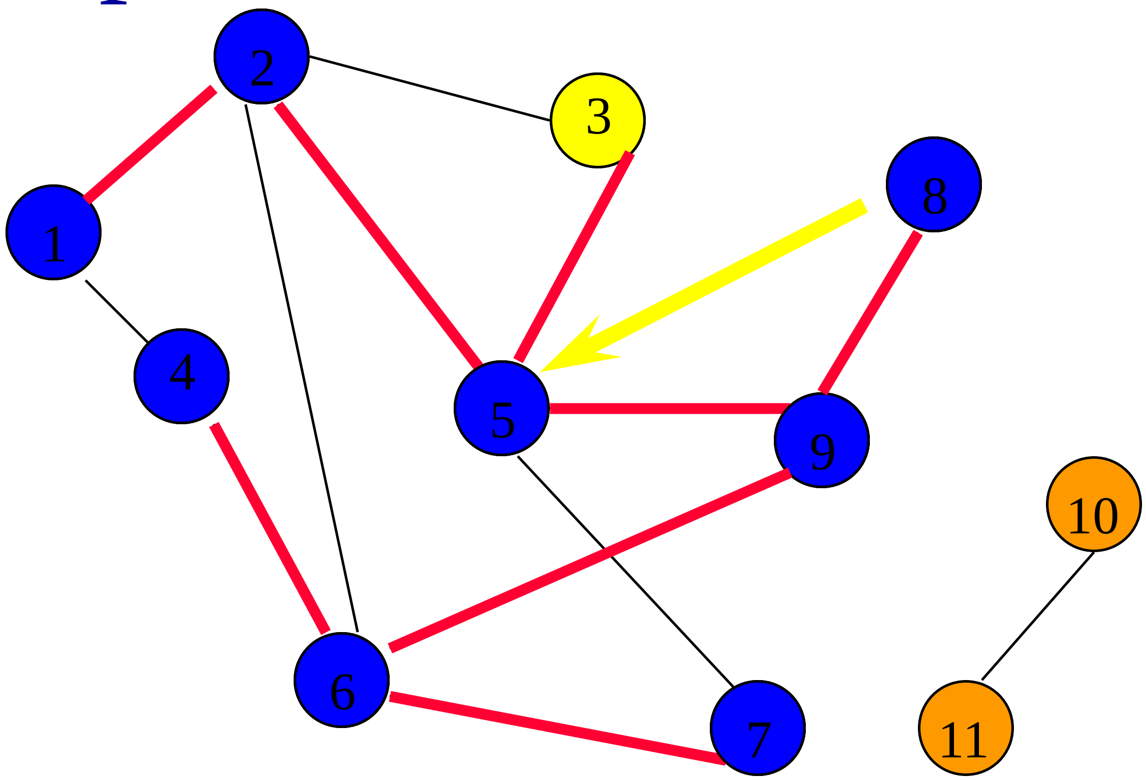
Return to 9.

Depth-First Search Example



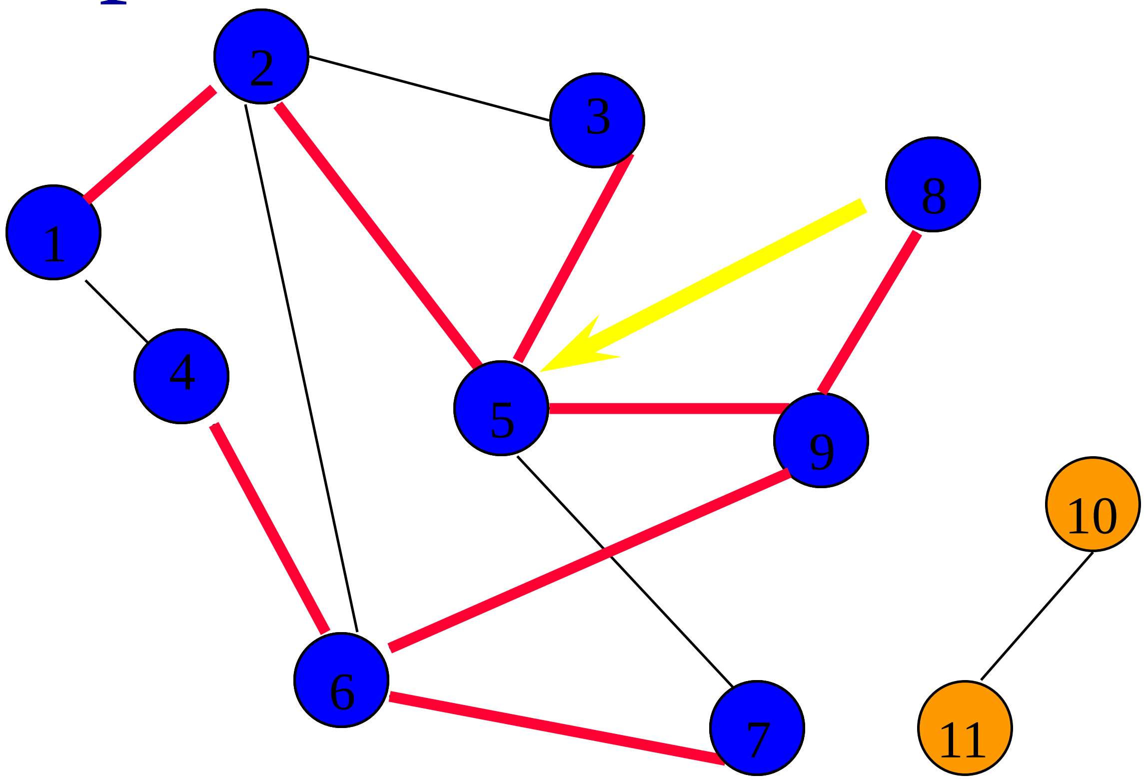
Return to 5.

Depth-First Search Example



Do a dfs(3).

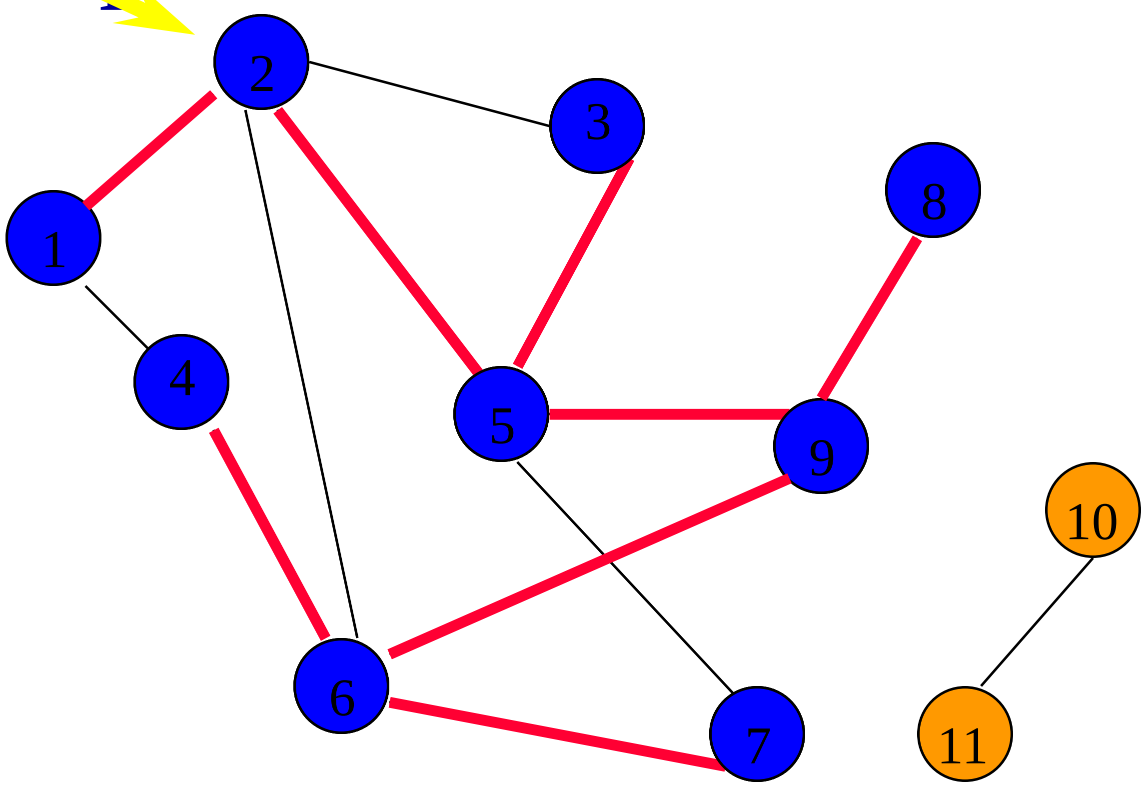
Depth-First Search Example



Label 3 and return to 5.

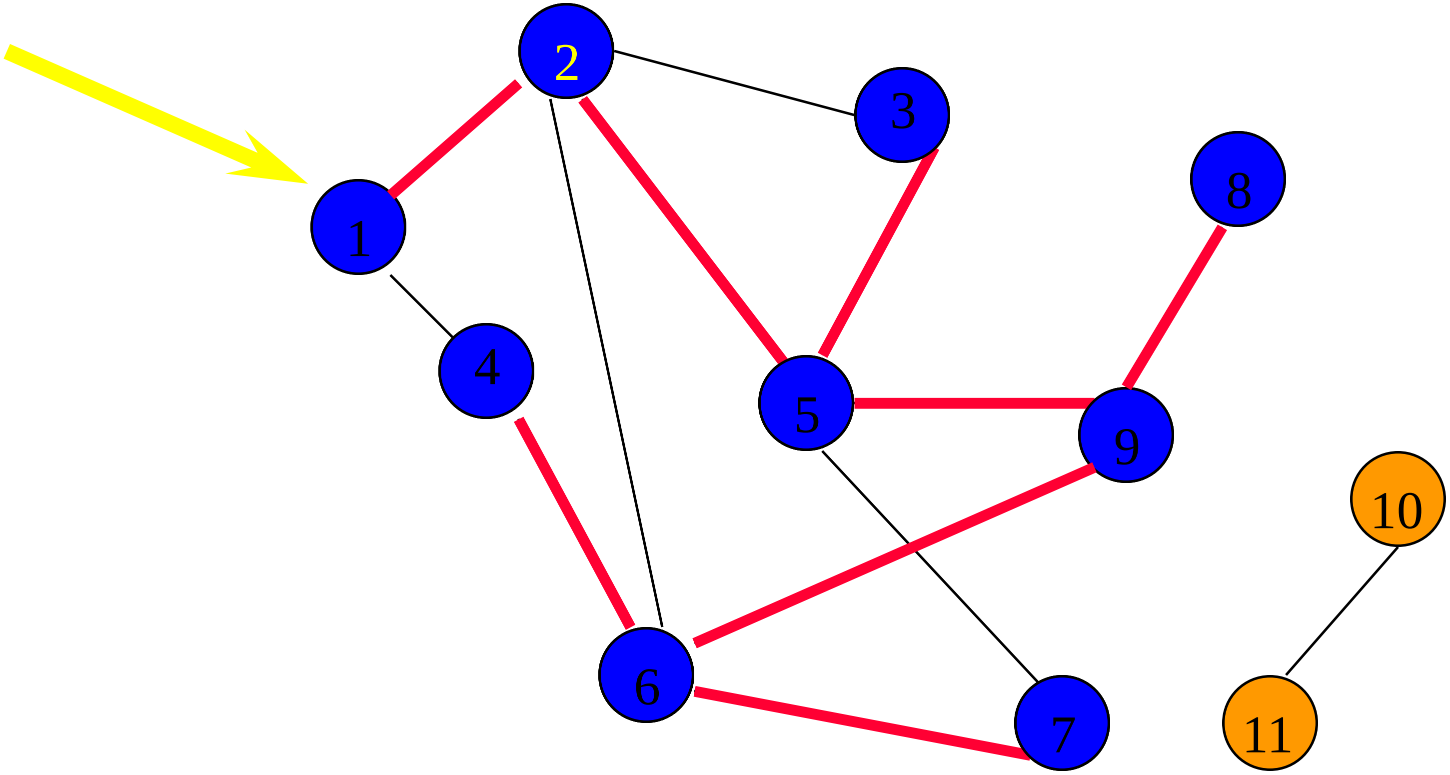
Return to 2.

Depth-First Search Example



Return to 1.

Depth-First Search Example



Return to invoking method.

Depth-First Search Property

- All vertices reachable from the start vertex (including the start vertex) are visited.

Path From Vertex v To Vertex u

- Start a depth-first search at vertex v.
- Terminate when vertex u is visited or when dfs ends (whichever occurs first).
- Time
  - $O(n^2)$ when adjacency matrix used
  - $O(n+e)$ when adjacency lists used (e is number of edges)

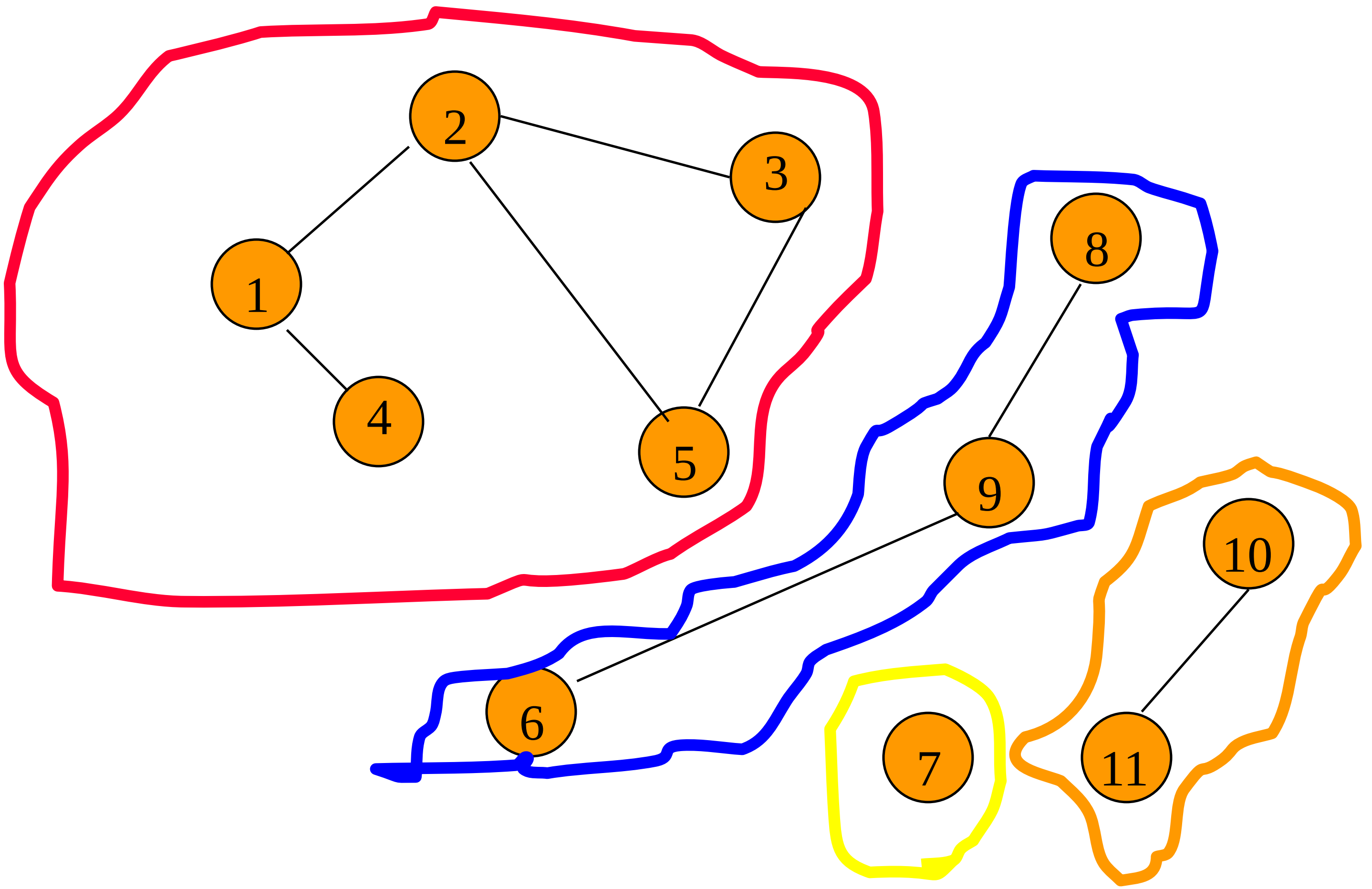
Is The Graph Connected?

- Start a depth-first search at any vertex of the graph.
- Graph is connected iff all n vertices get visited.
- Time
  - $O(n^2)$ when adjacency matrix used
  - $O(n+e)$ when adjacency lists used (e is number of edges)

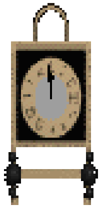
Connected Components

- Start a depth-first search at any as yet unvisited vertex of the graph.
- Newly visited vertices (plus edges between them) define a component.
- Repeat until all vertices are visited.

Connected Components

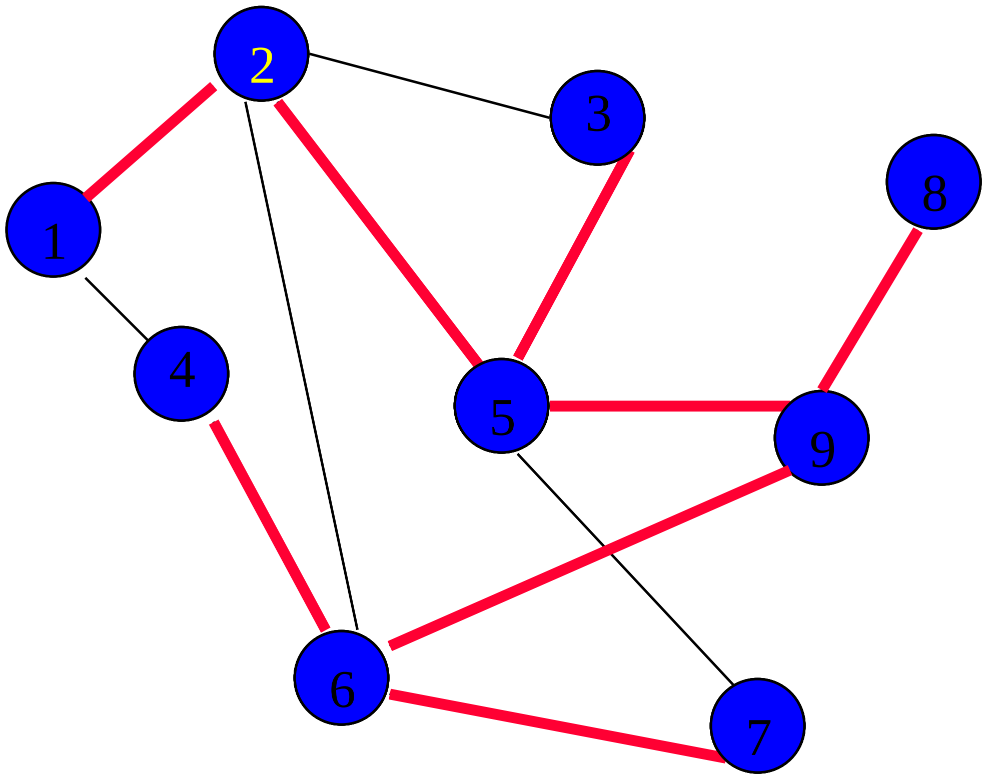


Time Complexity



- $O(n^2)$ when adjacency matrix used
- $O(n+e)$ when adjacency lists used (e is number of edges)

Spanning Tree



Depth-first search from vertex 1.

Depth-first spanning tree.

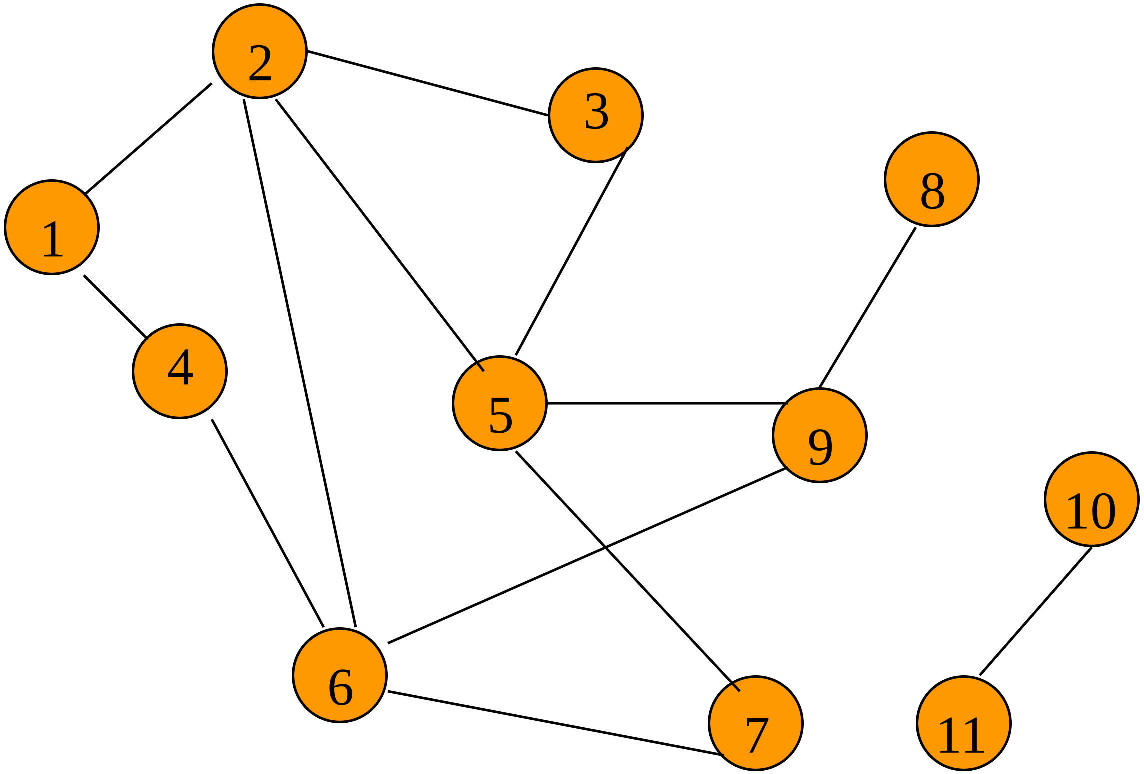
Spanning Tree

- Start a depth-first search at any vertex of the graph.
- If graph is connected, the n-1 edges used to get to unvisited vertices define a spanning tree (depth-first spanning tree).
- Time
  - $O(n^2)$ when adjacency matrix used
  - $O(n+e)$ when adjacency lists used (e is number of edges)

Breadth-First Search

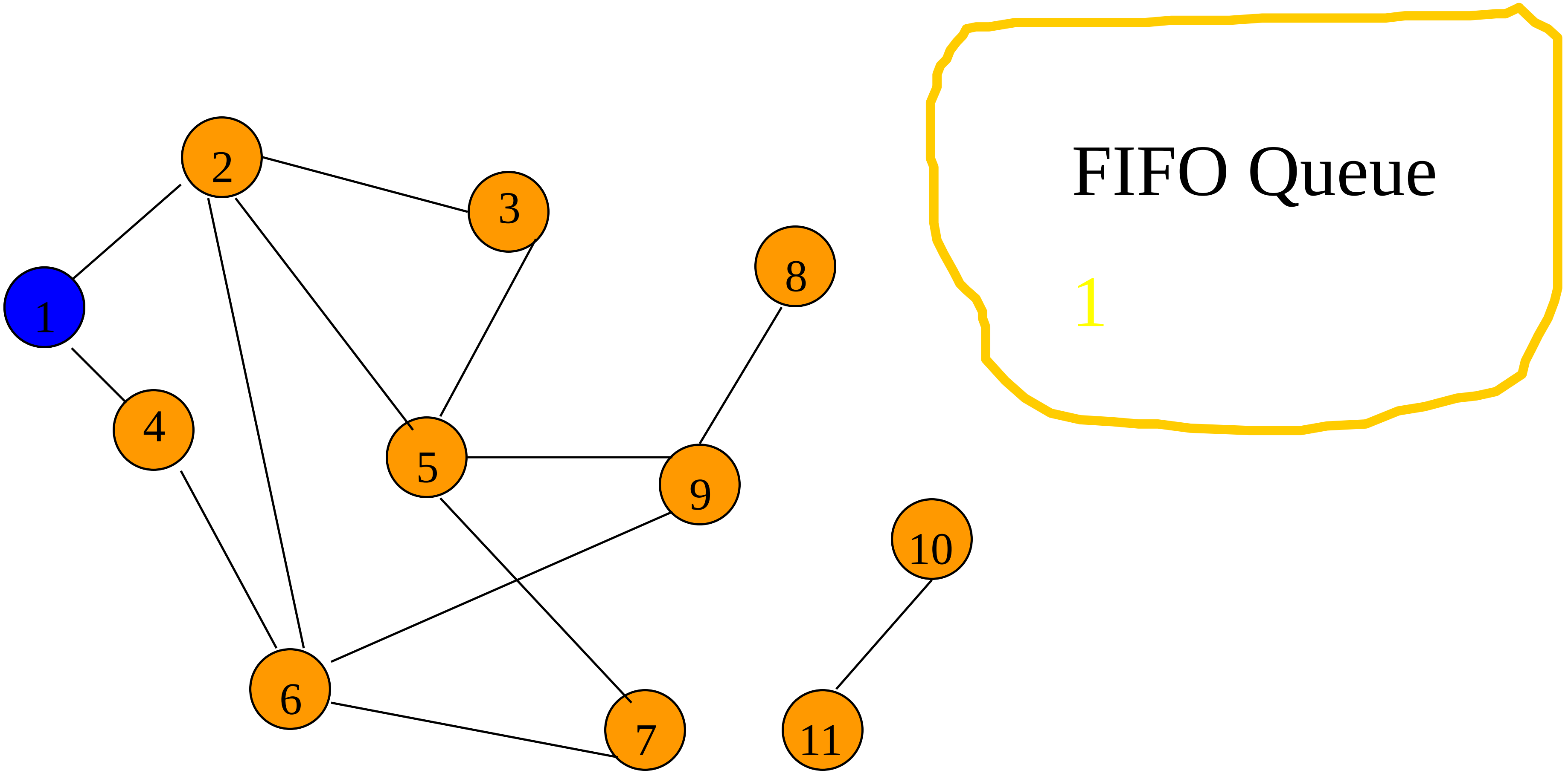
- Visit start vertex and put into a FIFO queue.
- Repeatedly remove a vertex from the queue, visit its unvisited adjacent vertices, put newly visited vertices into the queue.

Breadth-First Search Example



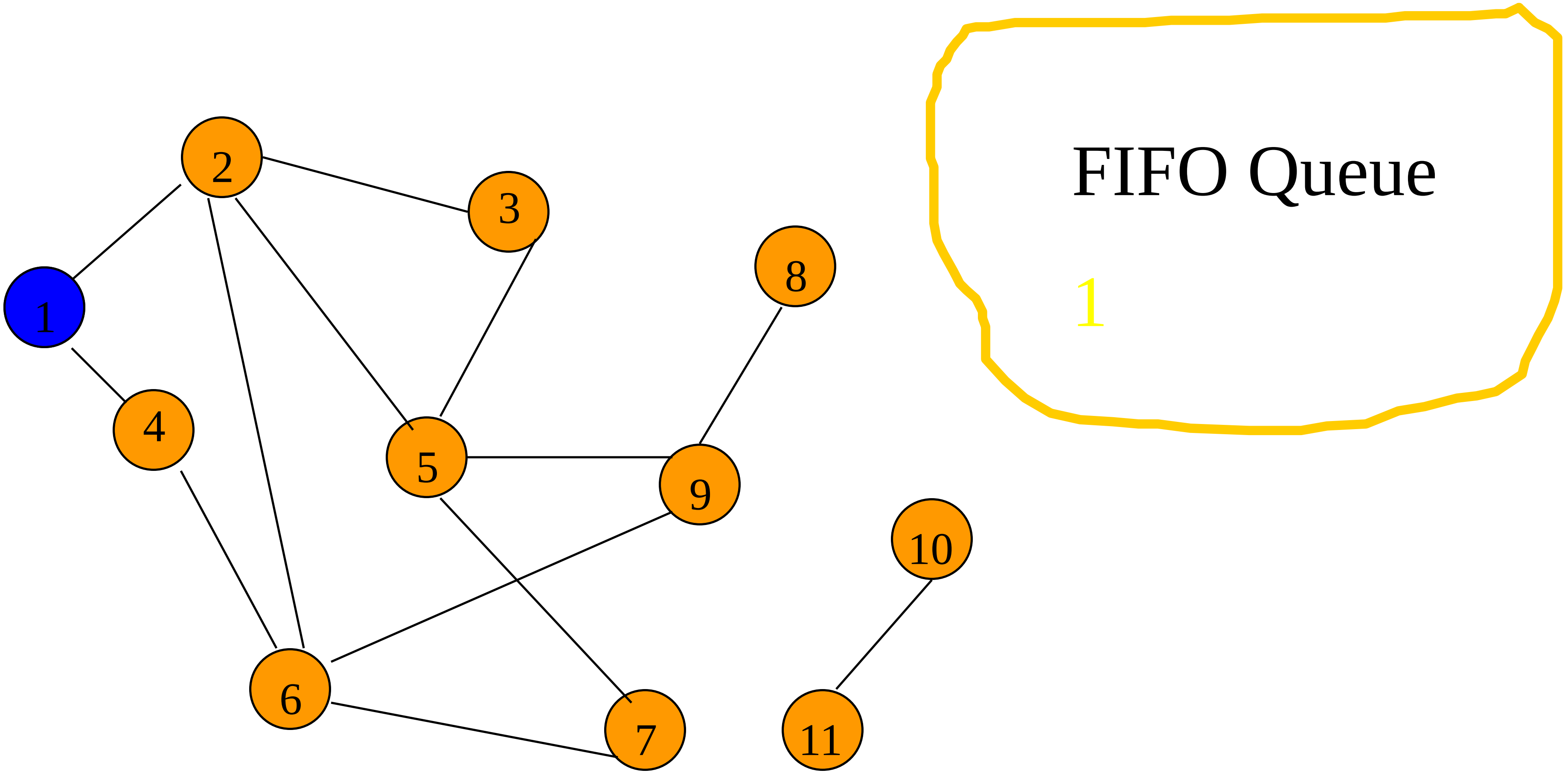
Start search at vertex 1.

Breadth-First Search Example



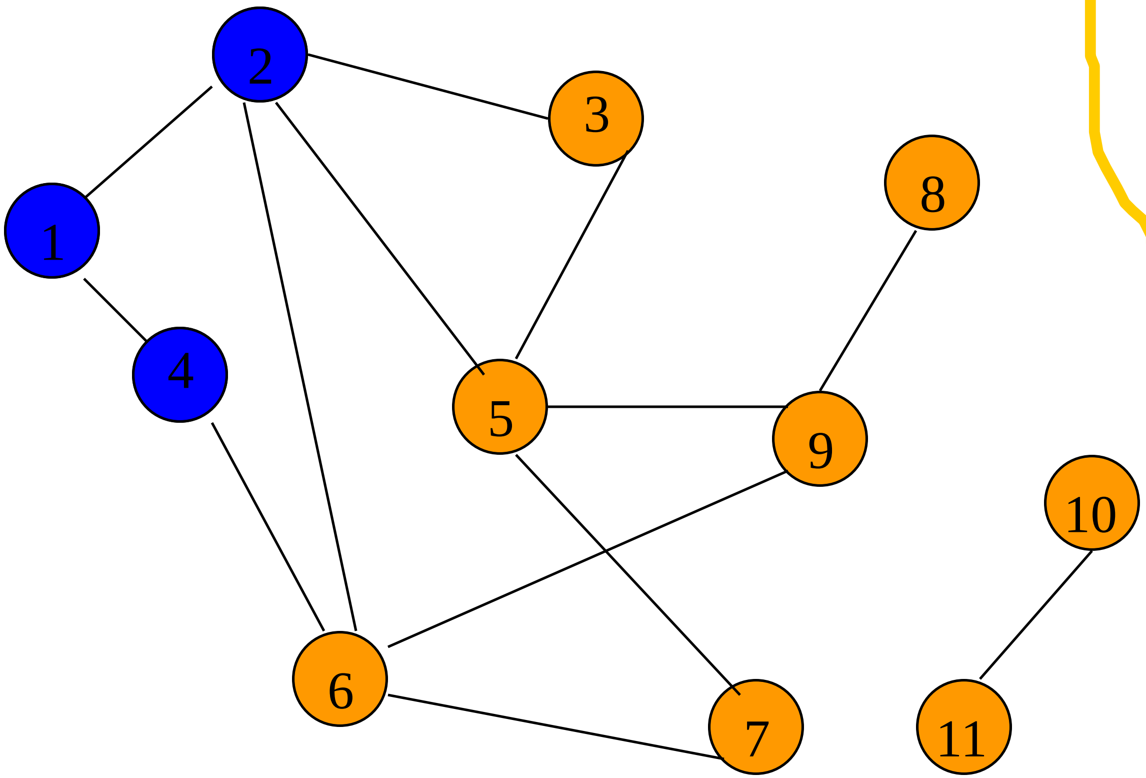
Visit/mark/label start vertex and put in a FIFO queue.

Breadth-First Search Example



Remove 1 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

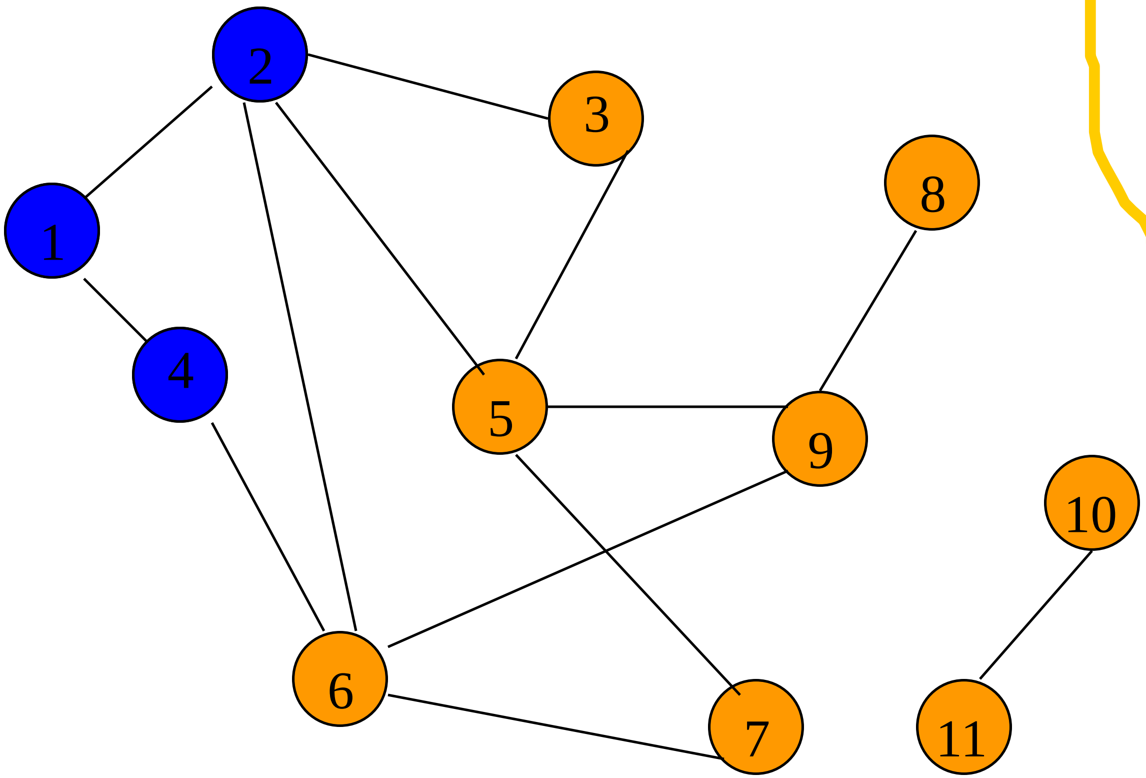


FIFO Queue

2 4

Remove 1 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

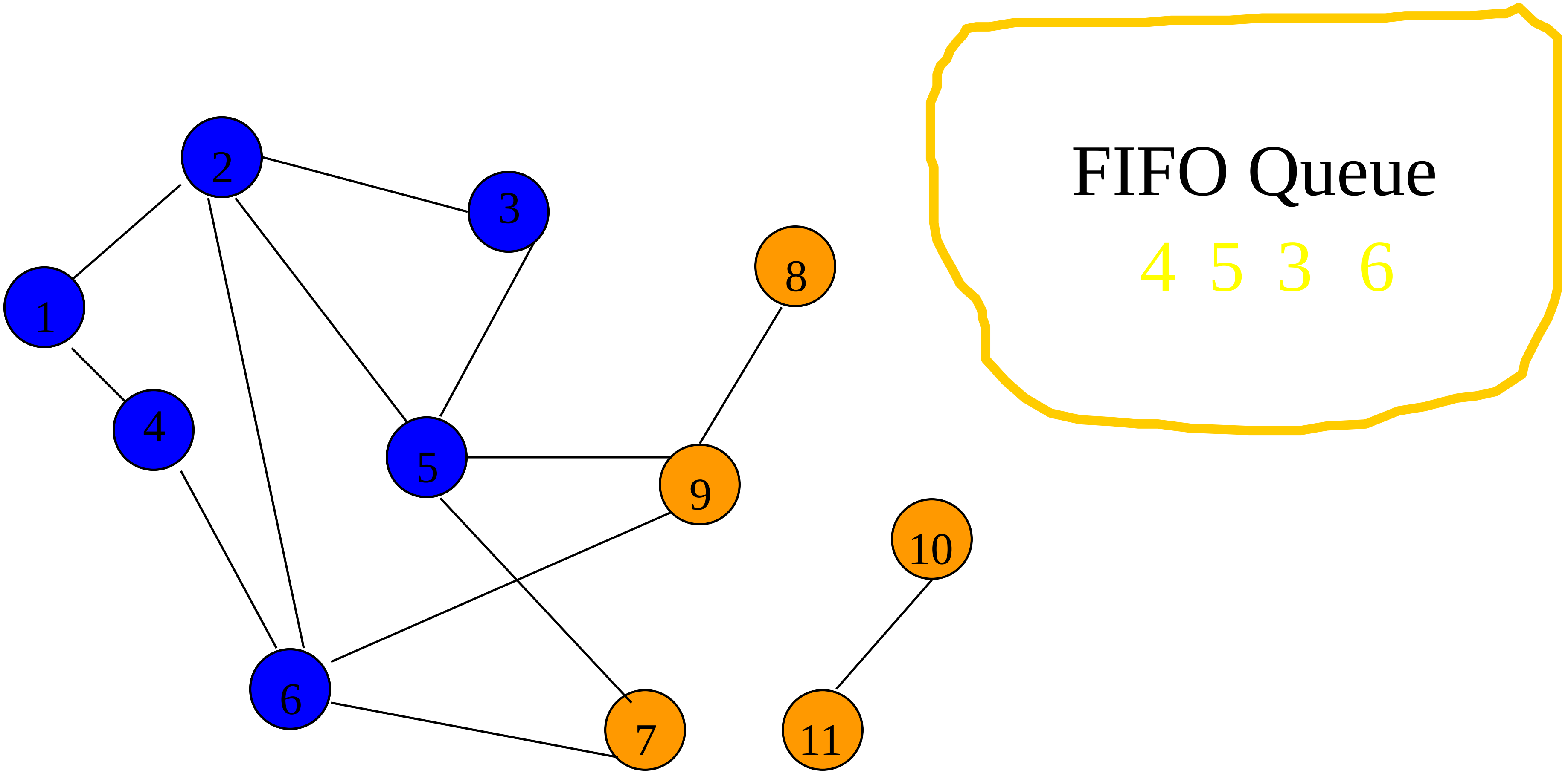


FIFO Queue

2 4

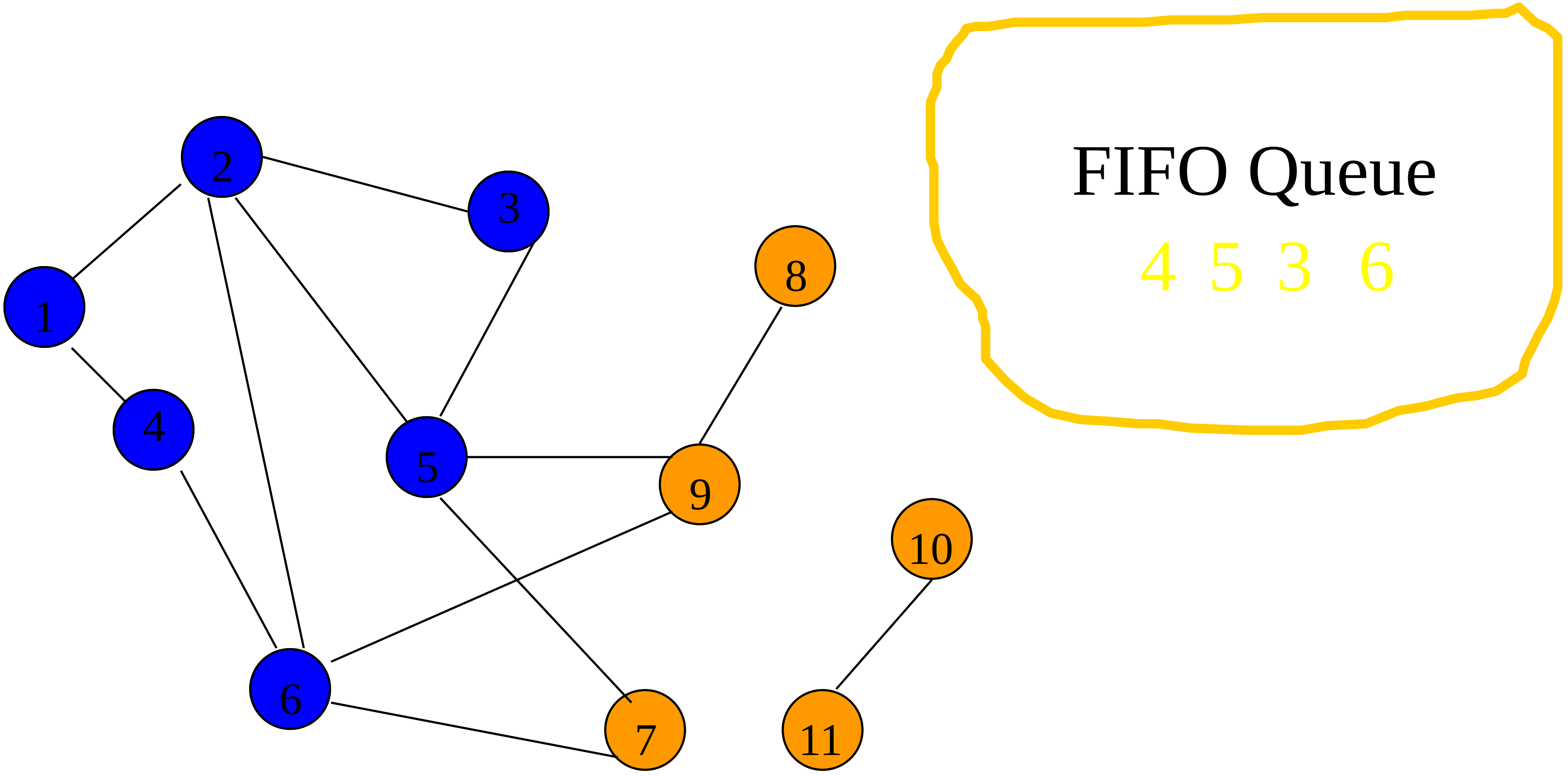
Remove 2 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



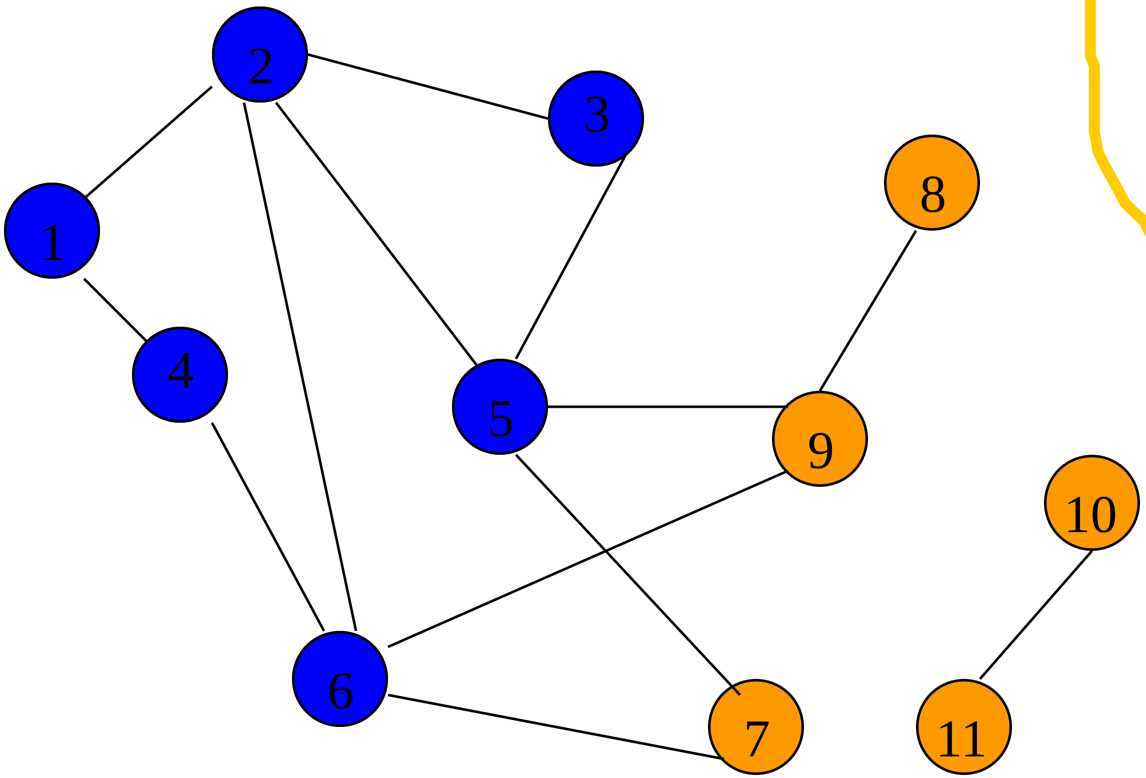
Remove 2 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



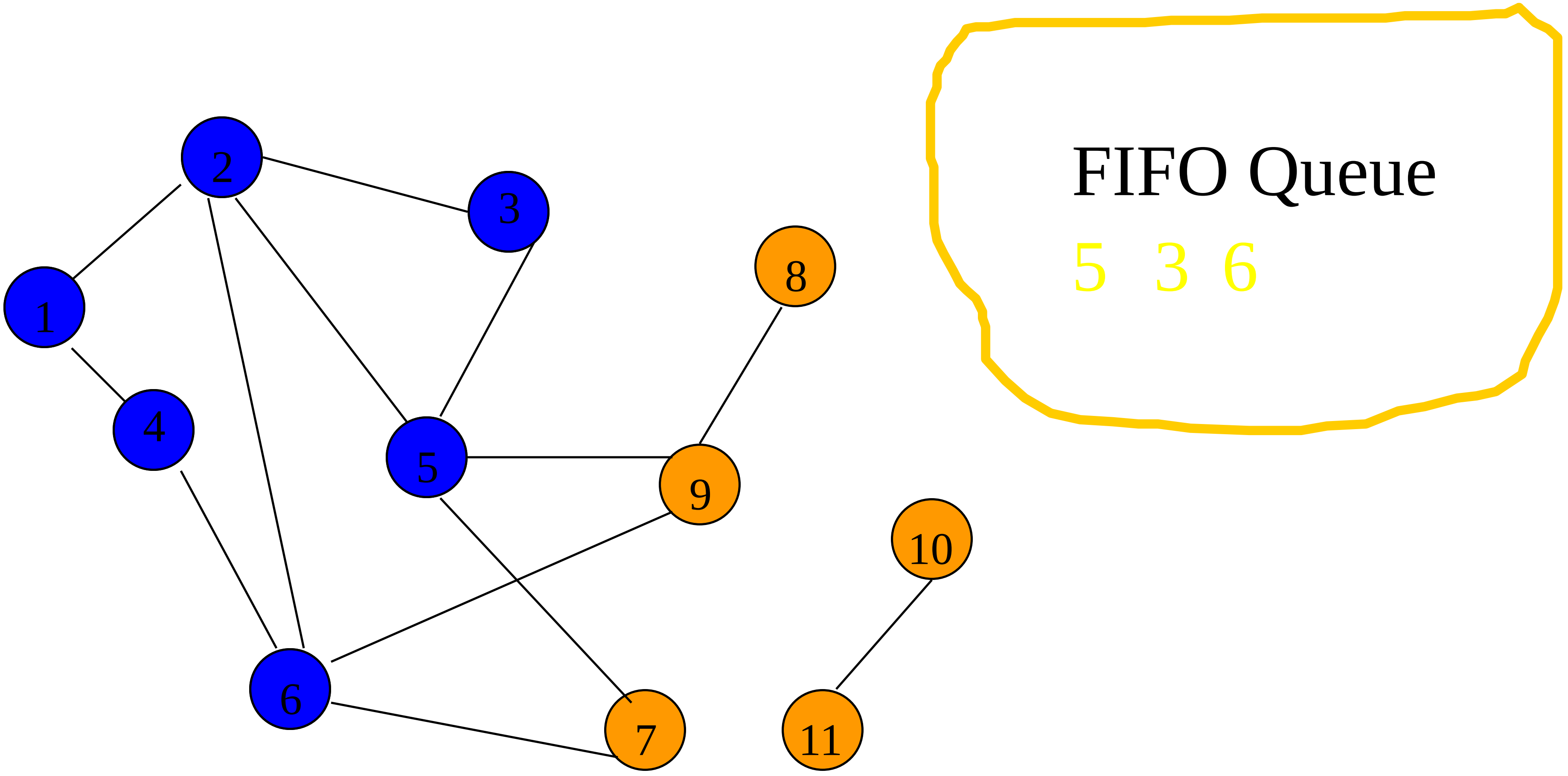
Remove 4 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



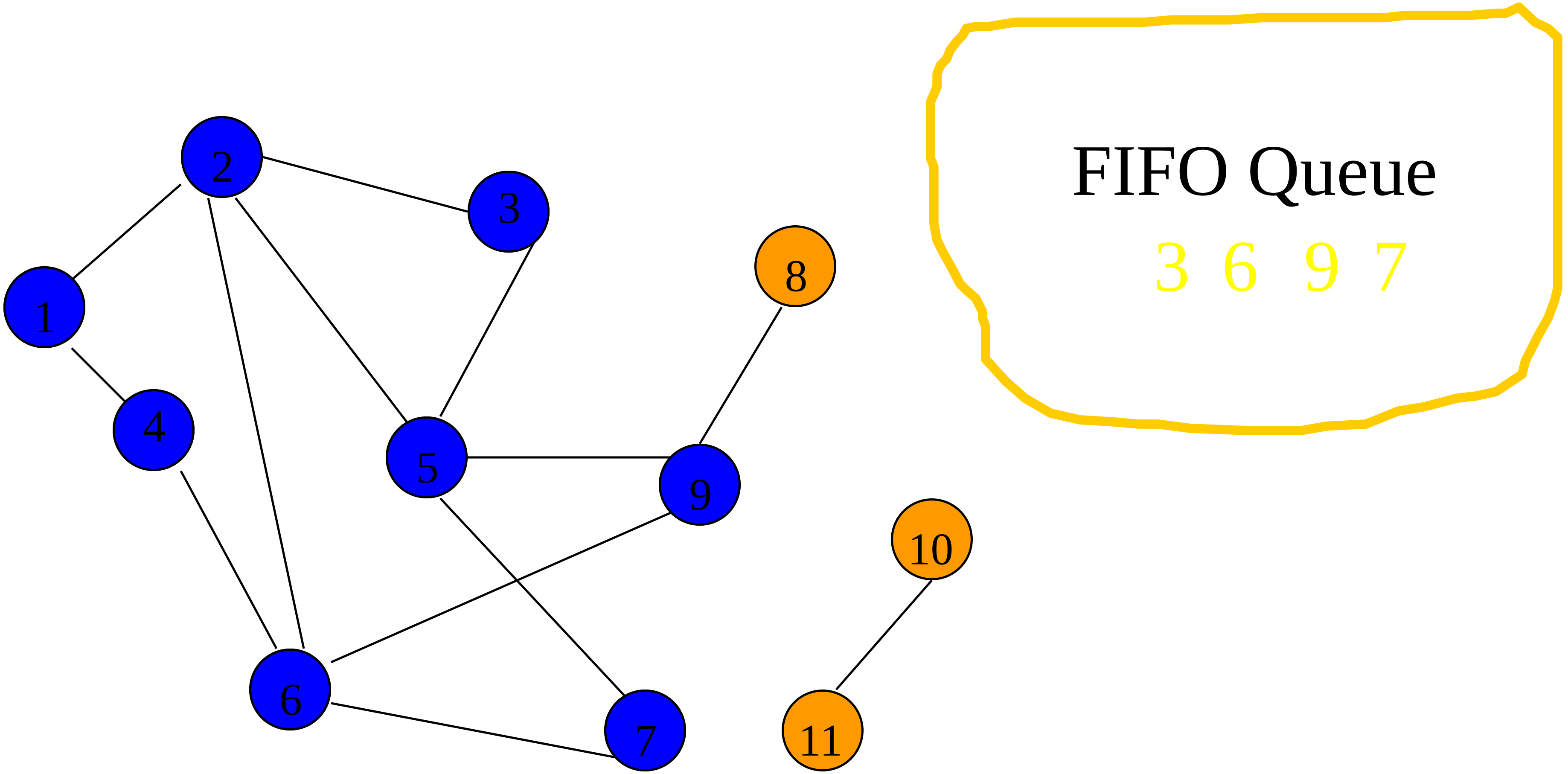
Remove 4 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



Remove 5 from Q; visit adjacent unvisited vertices; put in Q.

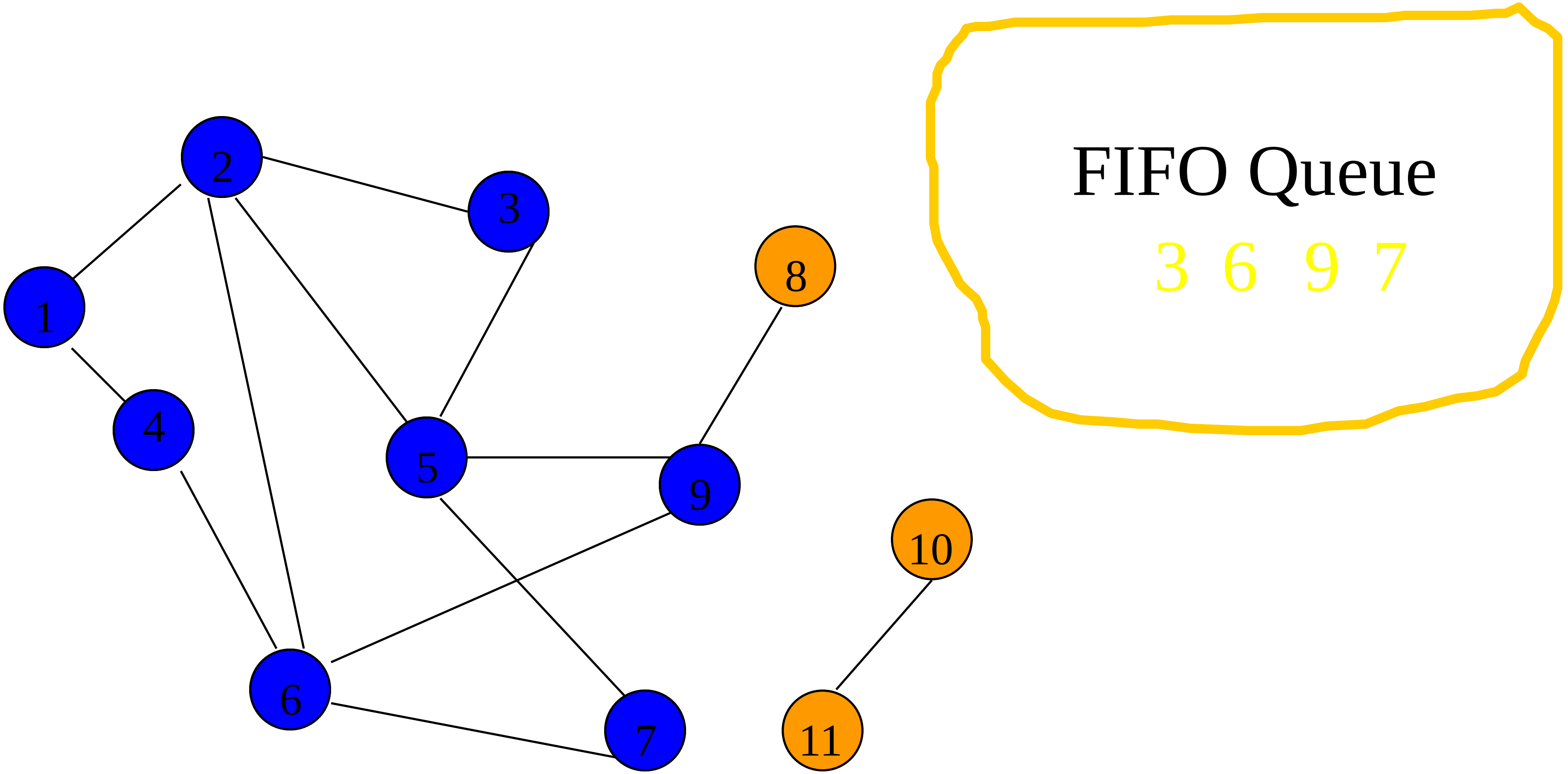
Breadth-First Search Example



Remove 5 from Q; visit adjacent unvisited vertices;

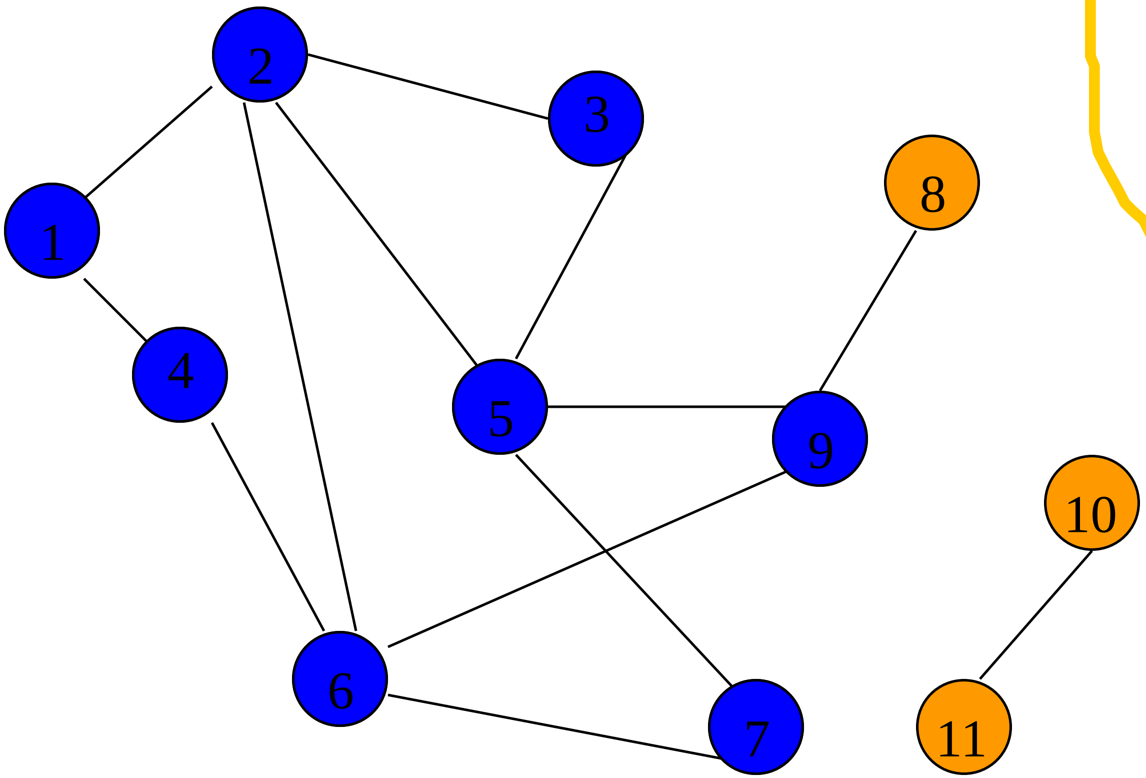
put in Q.

Breadth-First Search Example



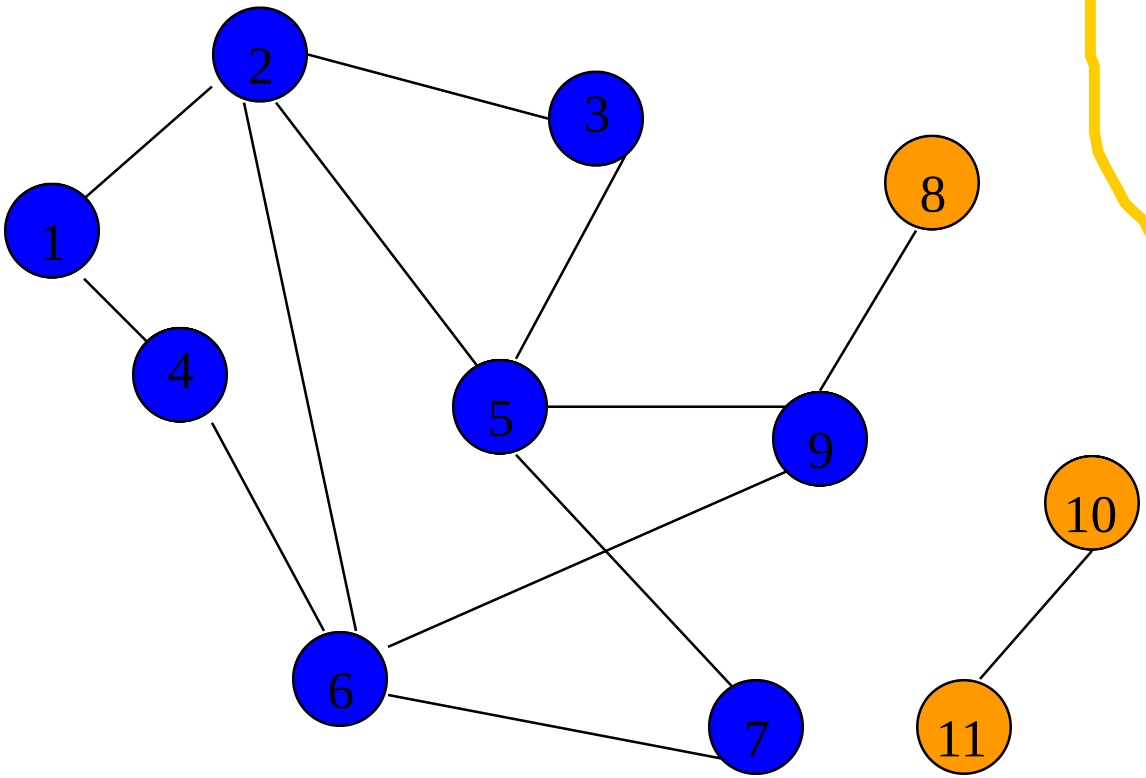
Remove 3 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



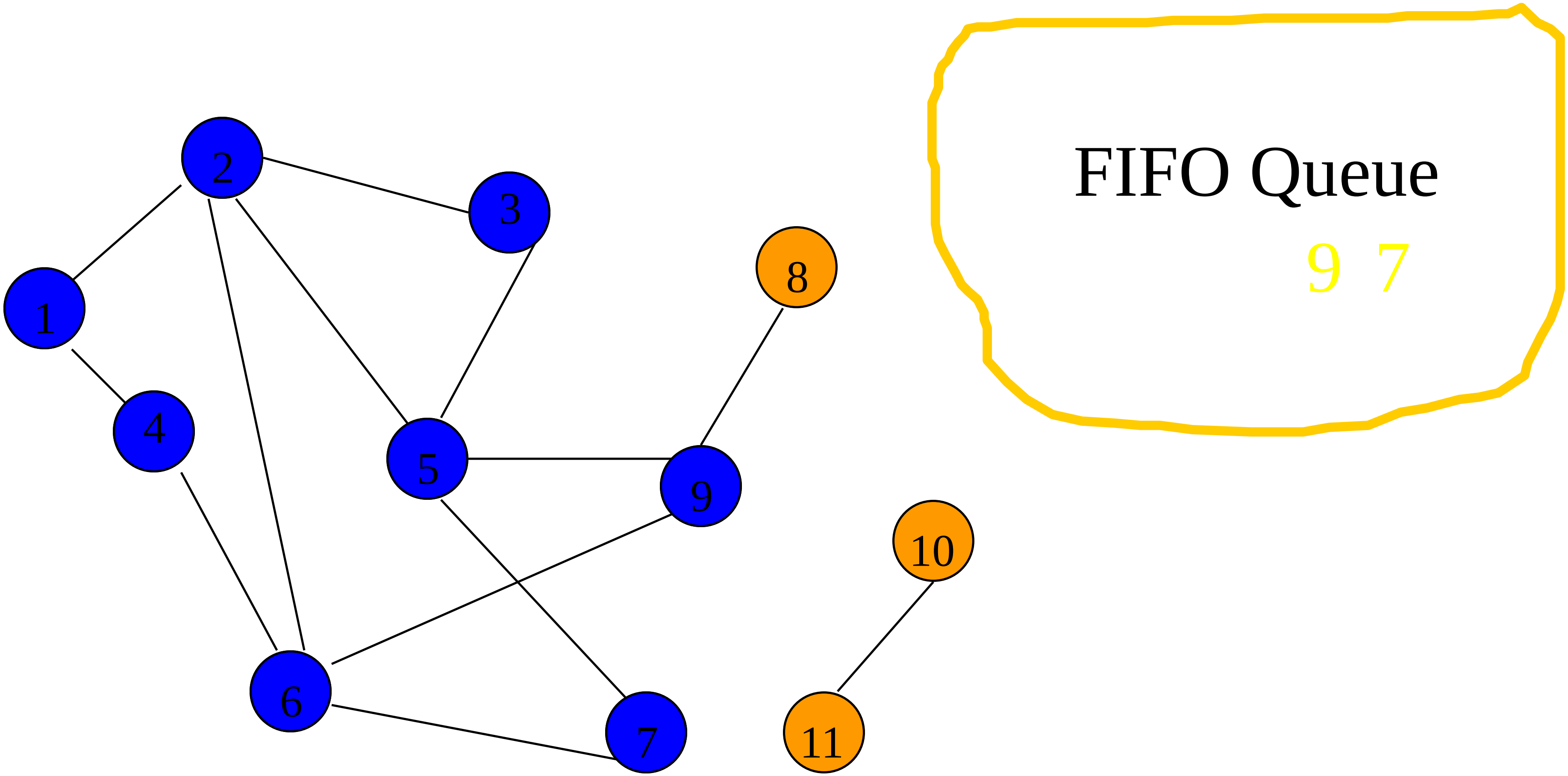
Remove 3 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



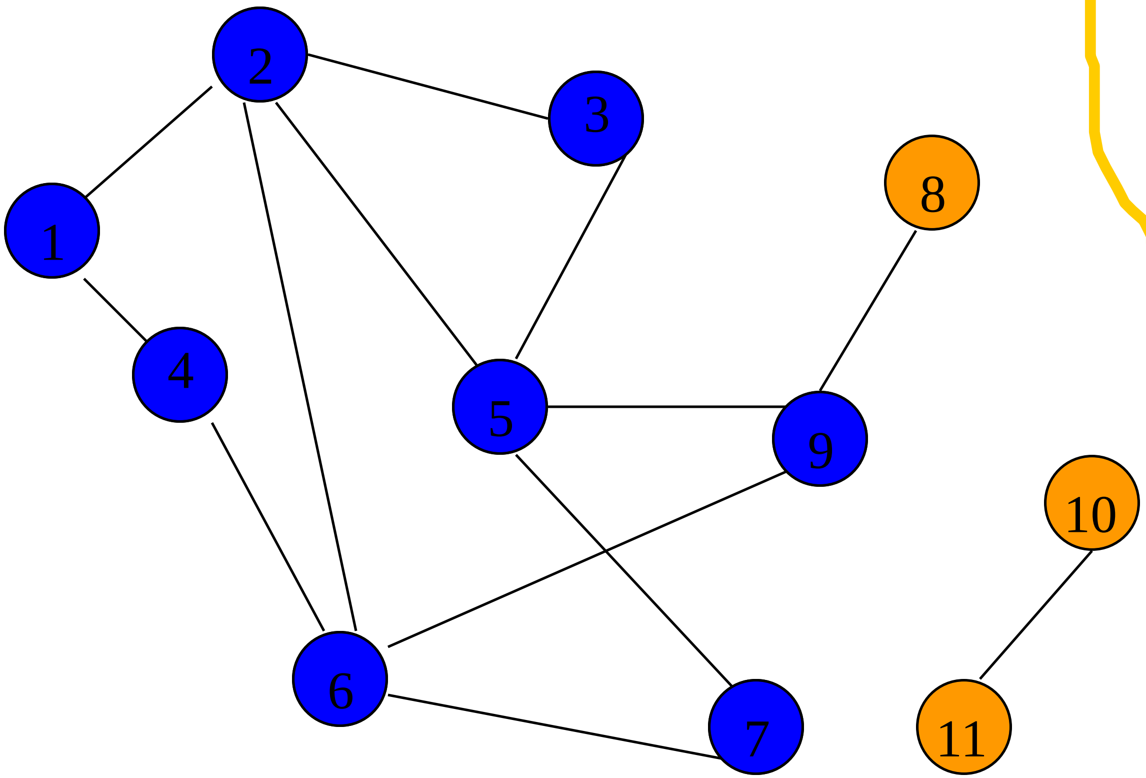
Remove 6 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



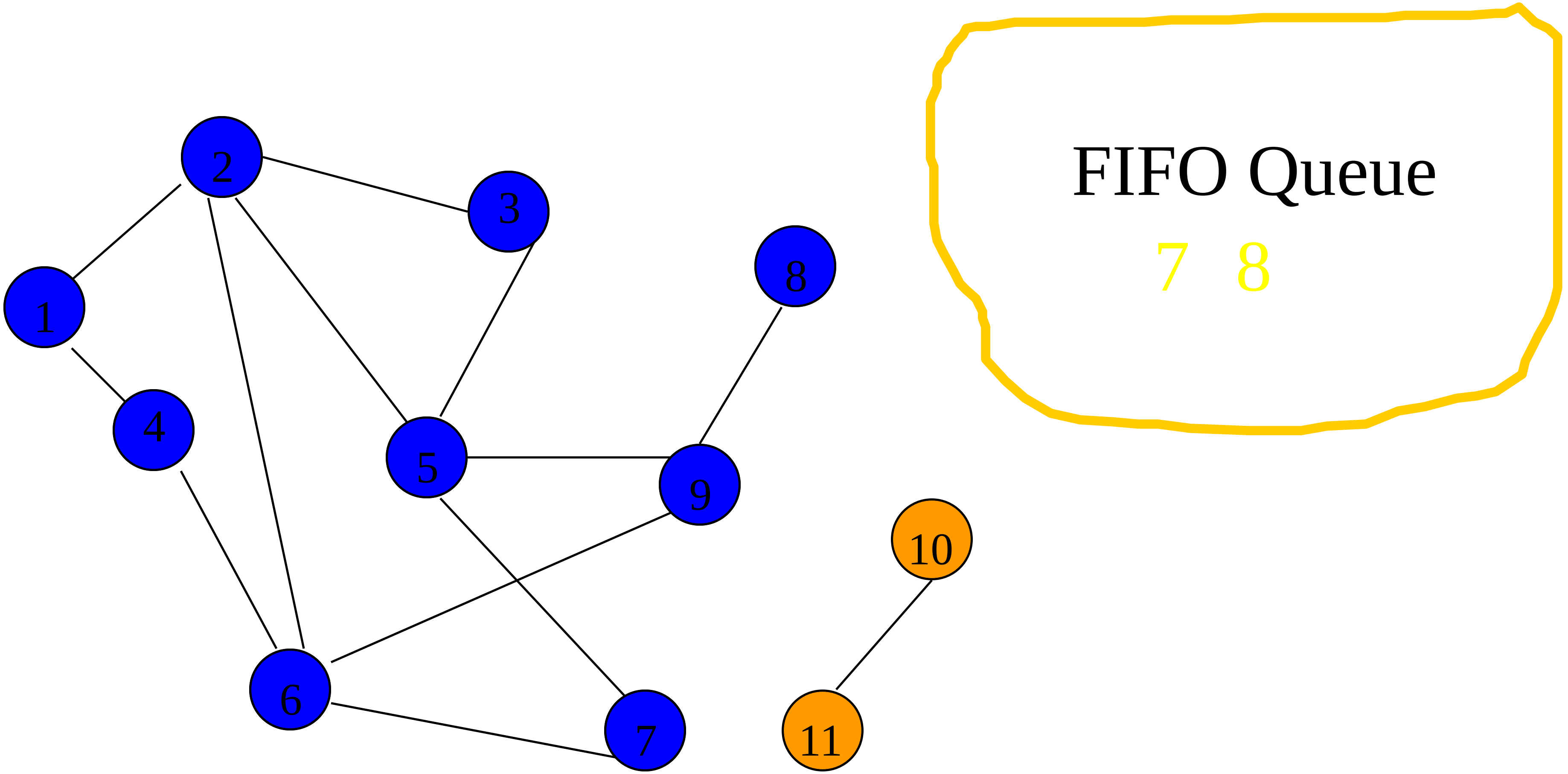
Remove 6 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example



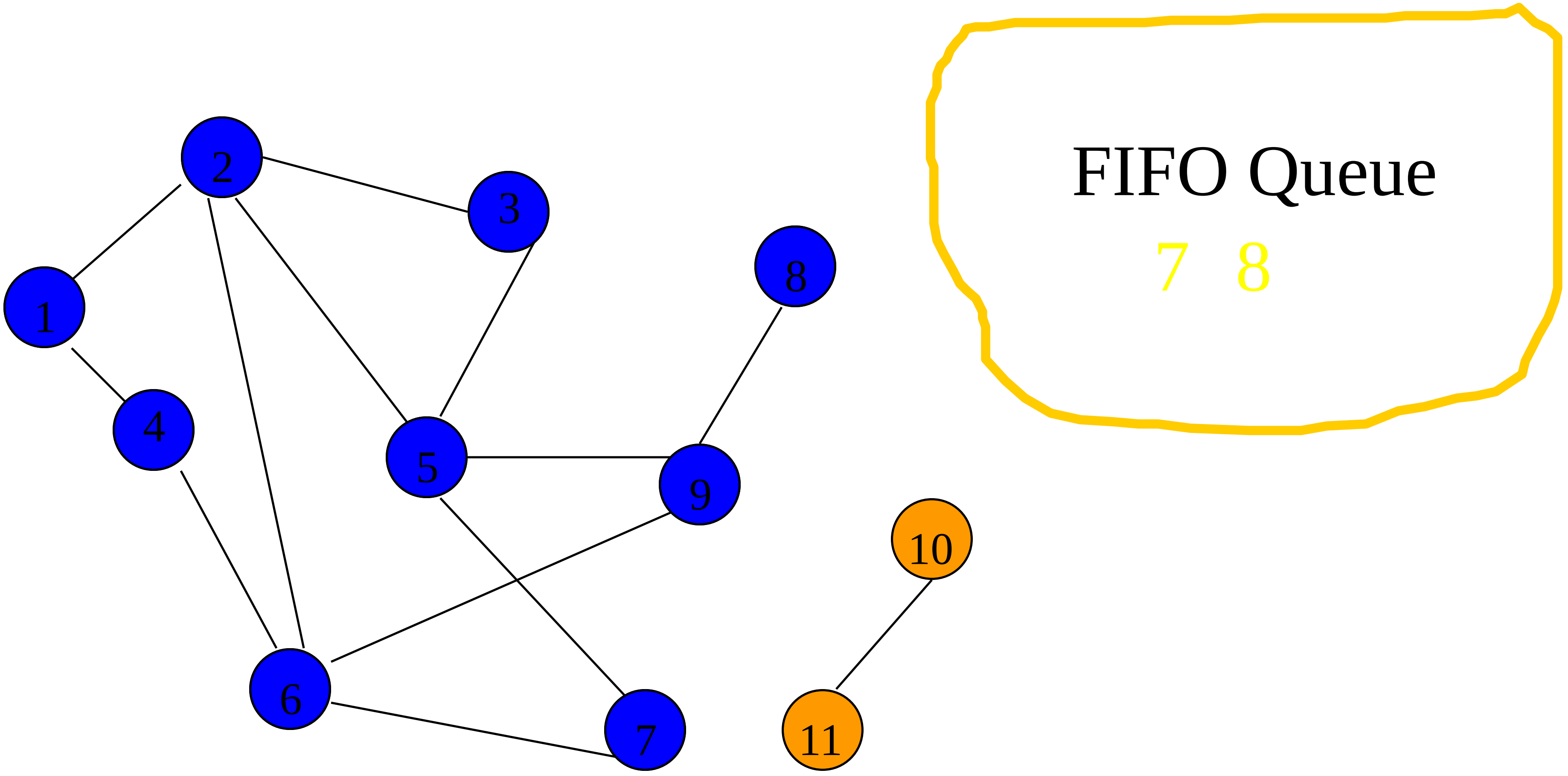
Remove 9 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



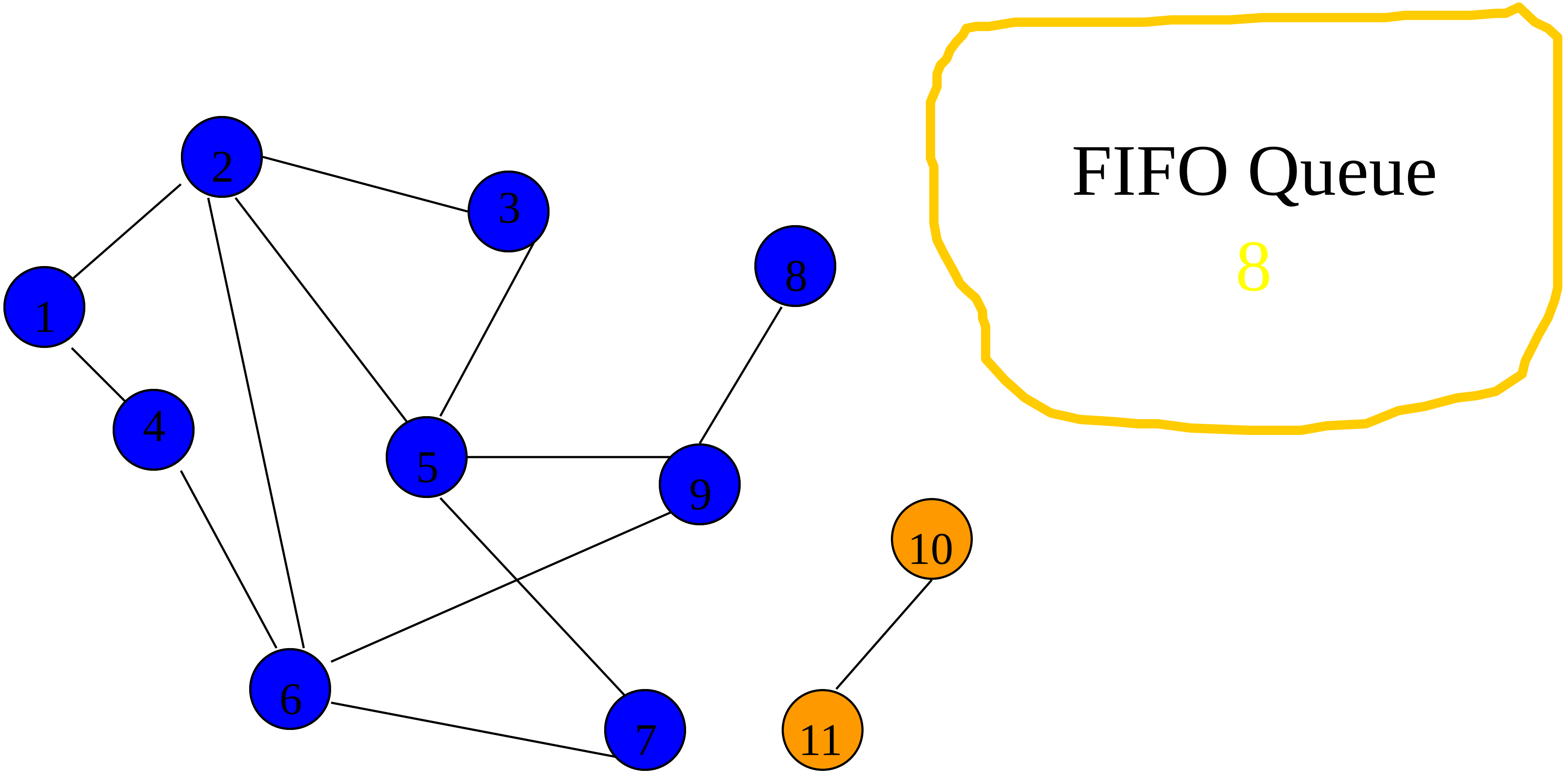
Remove 9 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example



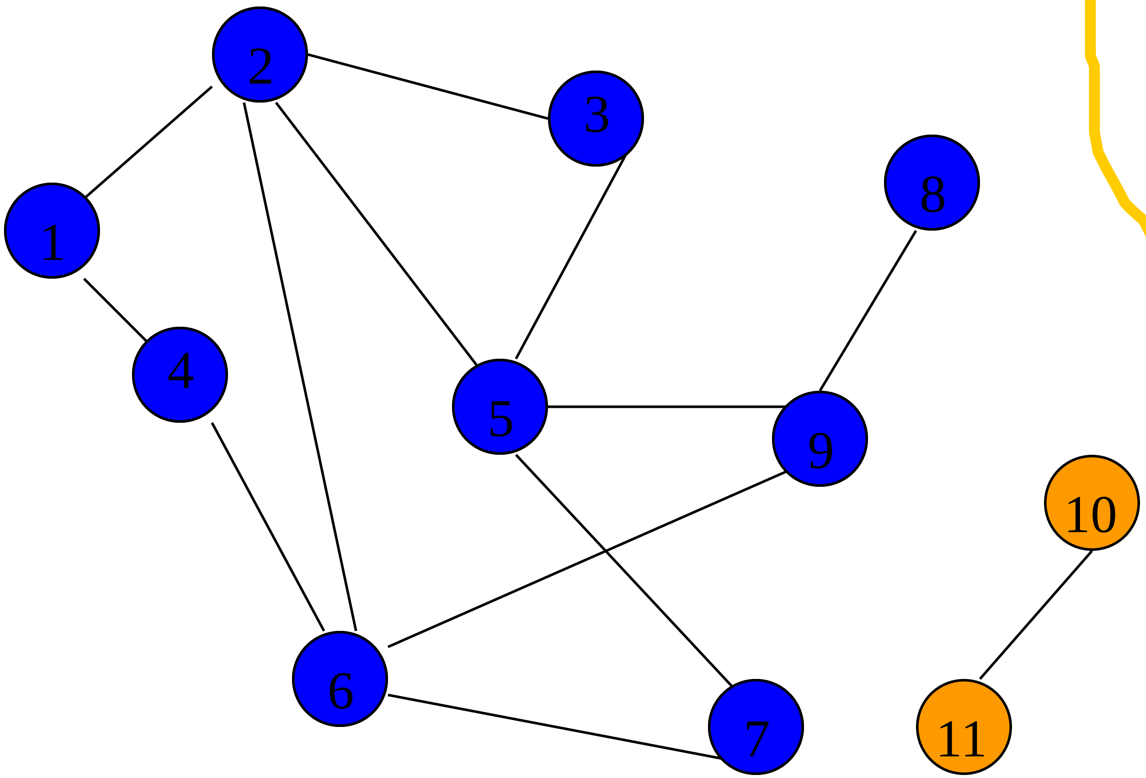
Remove 7 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example



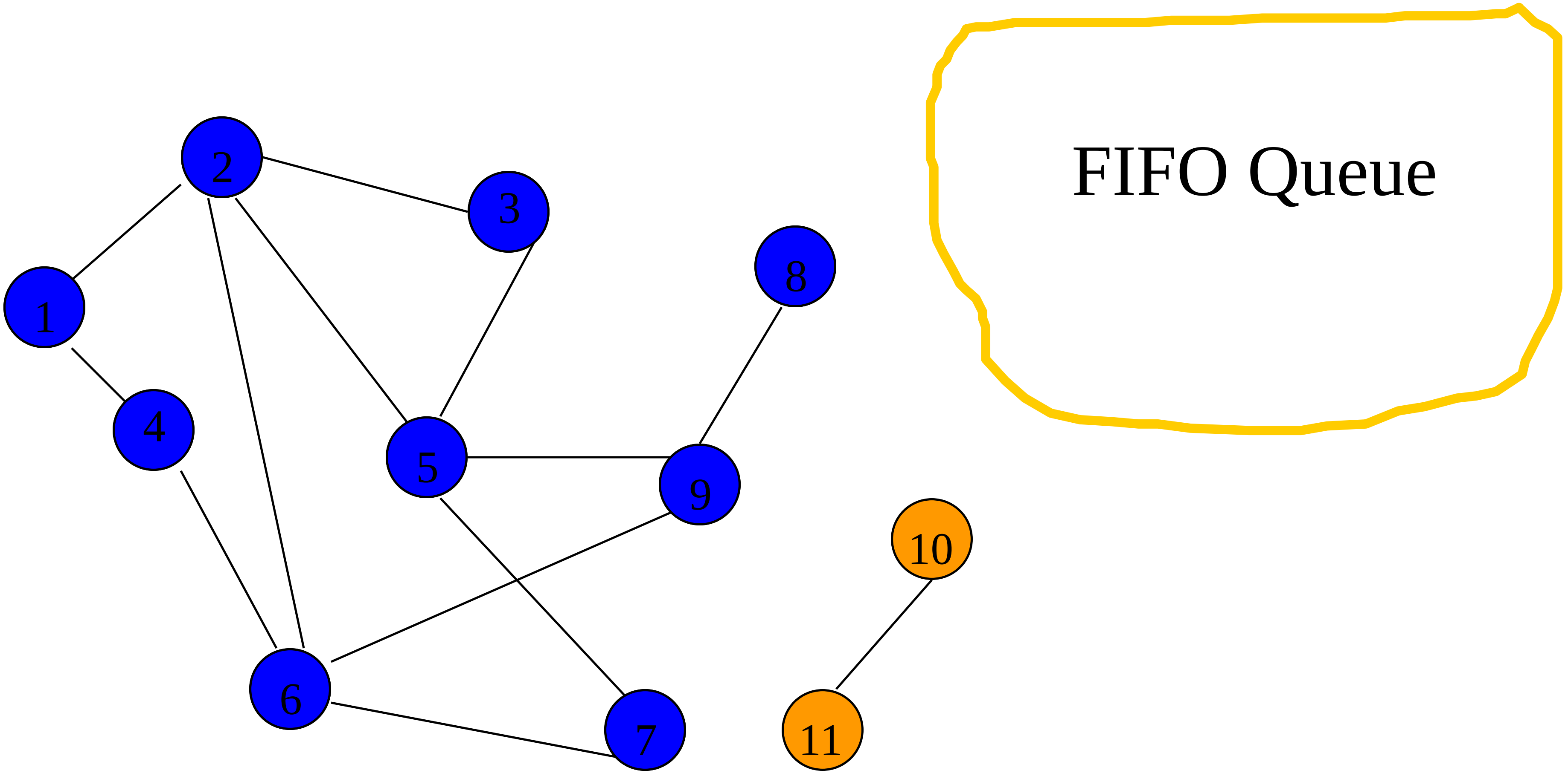
Remove 7 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



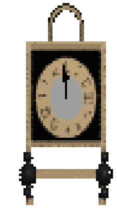
Remove 8 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example



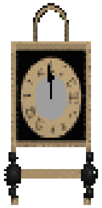
Queue is empty. Search terminates.

Time Complexity



- Each visited vertex is put on (and so removed from) the queue exactly once.
- When a vertex is removed from the queue, we examine its adjacent vertices.
  - O(n) if adjacency matrix used
  - O(vertex degree) if adjacency lists used
- Total time
  - O(mn), where m is number of vertices in the component that is searched (adjacency matrix)

Time Complexity



- O(n + sum of component vertex degrees) (adj. lists)

  = O(n + number of edges in component)

Breadth-First Search Properties

- Same complexity as dfs.
- Same properties with respect to path finding, connected components, and spanning trees.
- Edges used to reach unlabeled vertices define a depth-first spanning tree when the graph is connected.
- There are problems for which bfs is better than dfs and vice versa.

Disjoint Sets

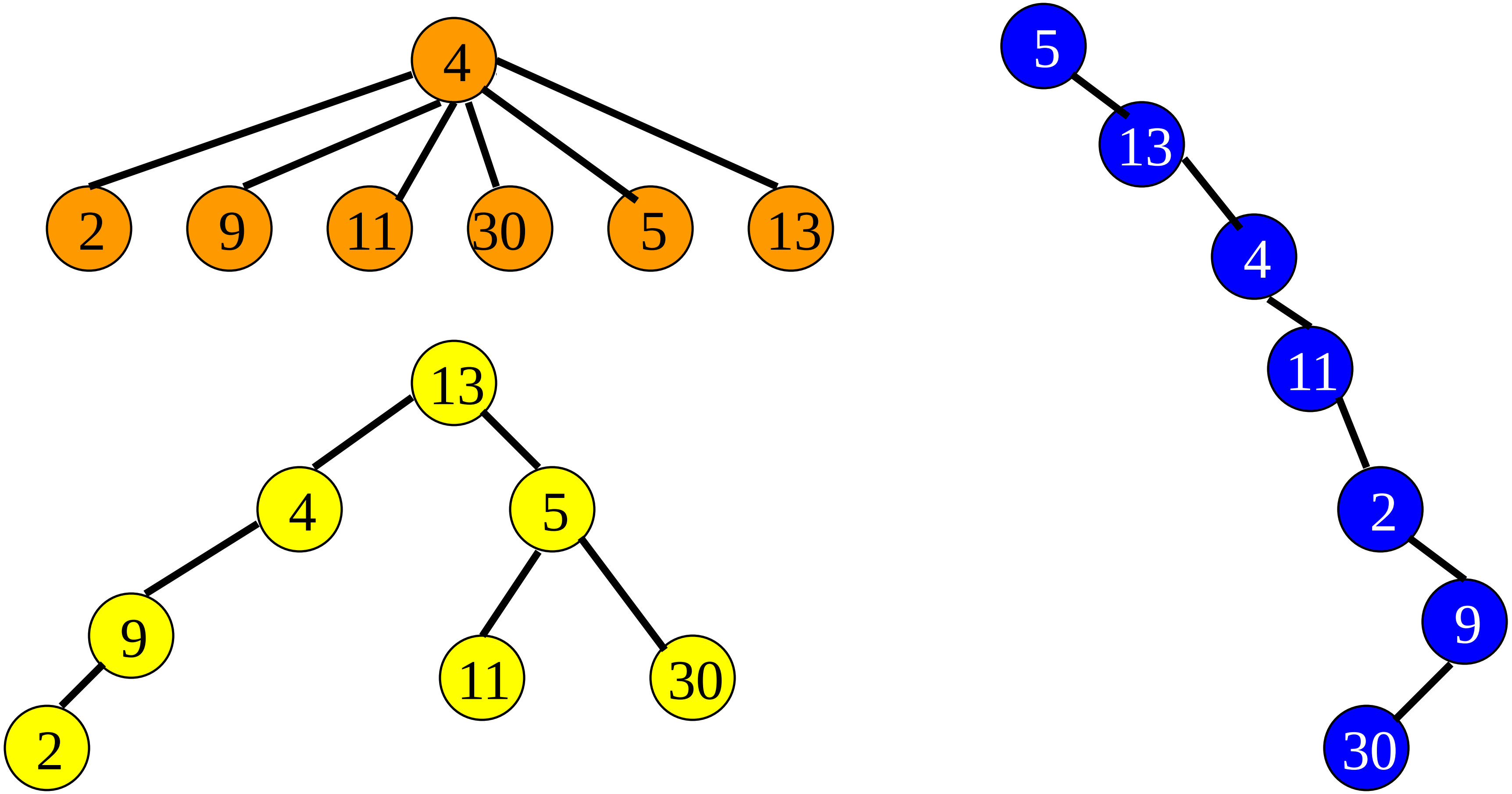
- Given a set {1, 2, …, n} of n elements.
- Initially each element is in a different set.
  - {1}, {2}, …, {n}
- An intermixed sequence of union and find operations is performed.
- A union operation combines two sets into one.
  - Each of the n elements is in exactly one set at any time.
- A find operation identifies the set that contains a particular element.

Using Arrays And Chains

- Best time complexity using arrays and chains is $O(n + u \log u + f)$, where u and f are, respectively, the number of union and find operations that are done.
- Using a tree (not a binary tree) to represent a set, the time complexity becomes almost $O(n + f)$ (assuming at least $n/2$ union operations).

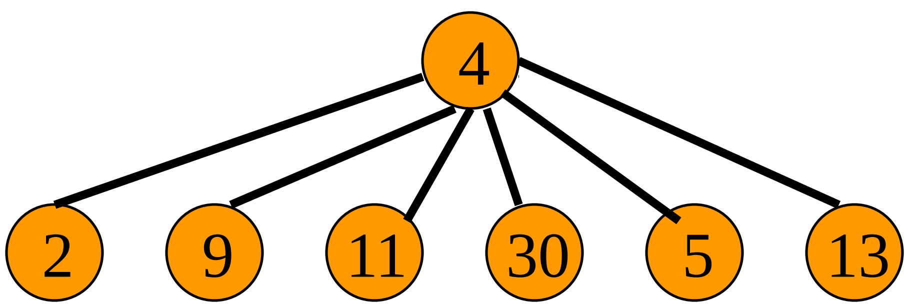
A Set As A Tree

- S = {2, 4, 5, 9, 11, 13, 30}
- Some possible tree representations:



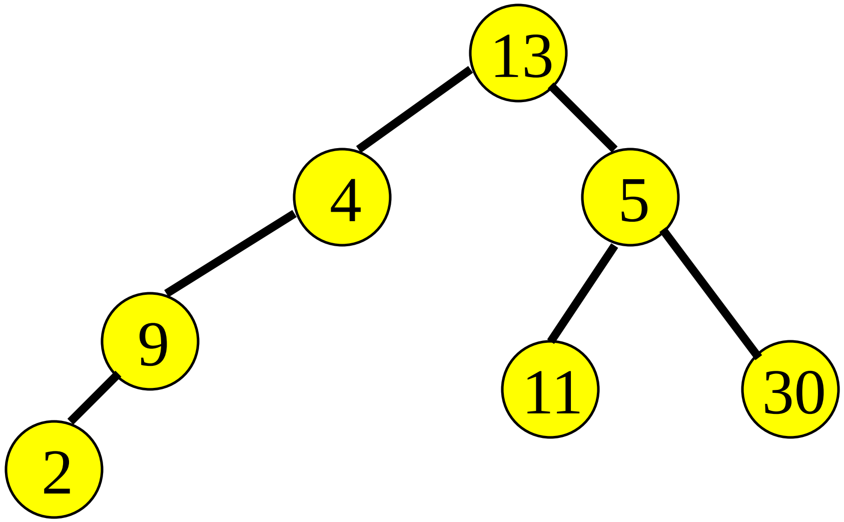
Result Of A Find Operation

- find(i) is to identify the set that contains element i.
- In most applications of the union-find problem, the user does not provide set identifiers.
- The requirement is that find(i) and find(j) return the same value iff elements i and j are in the same set.



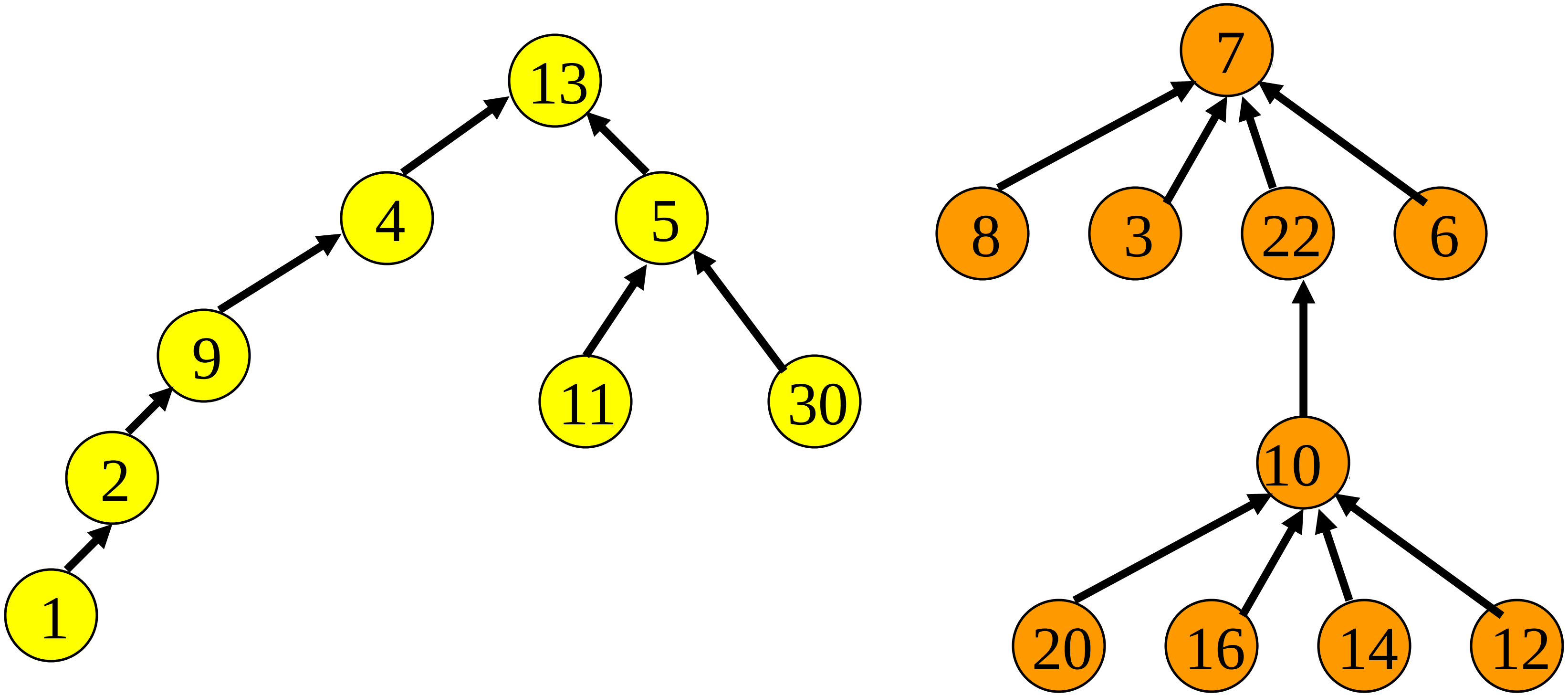
find(i) will return the element that is in the tree root.

Strategy For find(i)



- Start at the node that represents element i and climb up the tree until the root is reached.
- Return the element in the root.
- To climb the tree, each node must have a parent pointer.

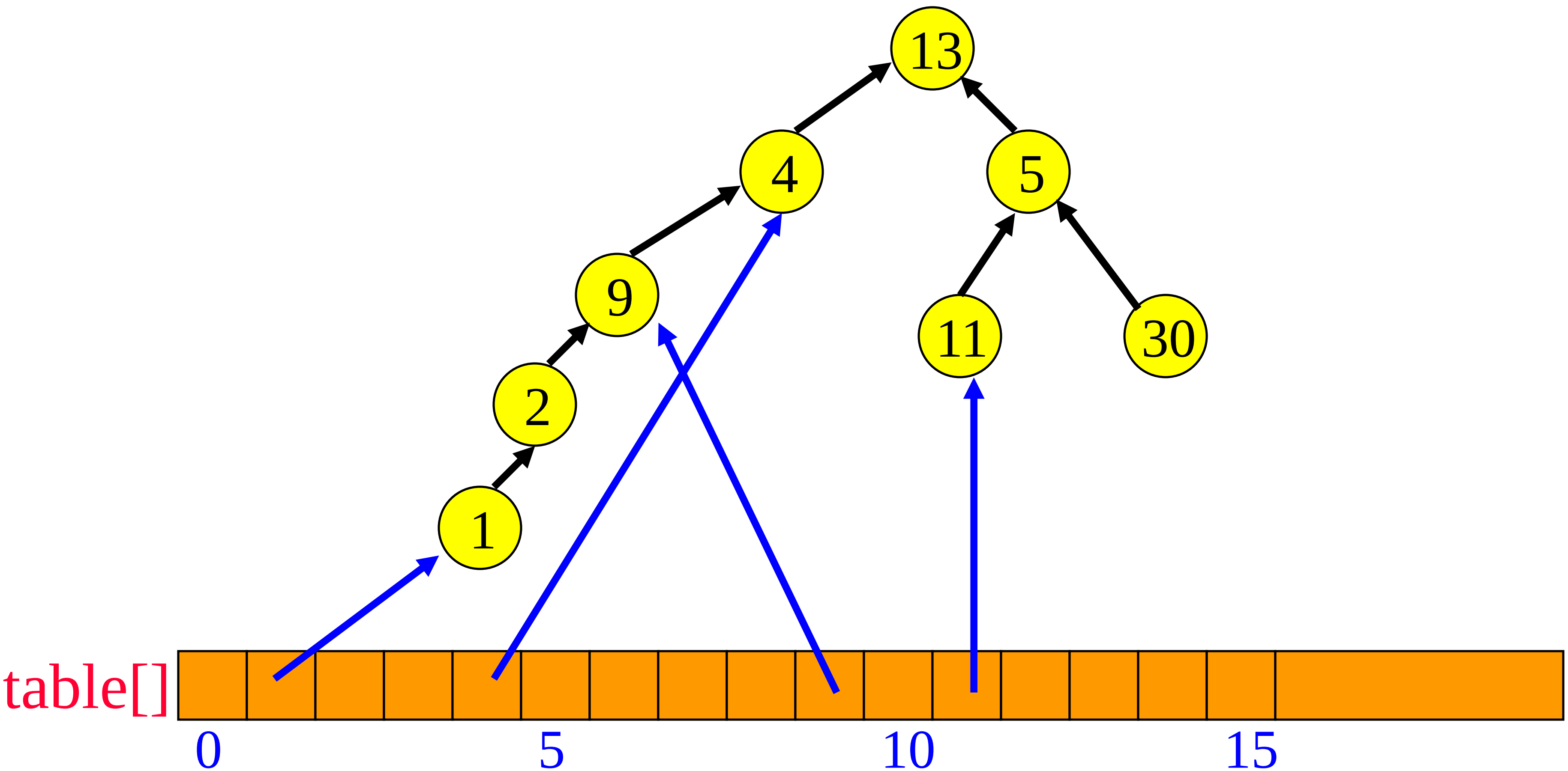
Trees With Parent Pointers



Possible Node Structure

- Use nodes that have two fields: element and parent.
  - Use an array table[] such that table[i] is a pointer to the node whose element is i.
  - To do a find(i) operation, start at the node given by table[i] and follow parent fields until a node whose parent field is null is reached.
  - Return element in this root node.

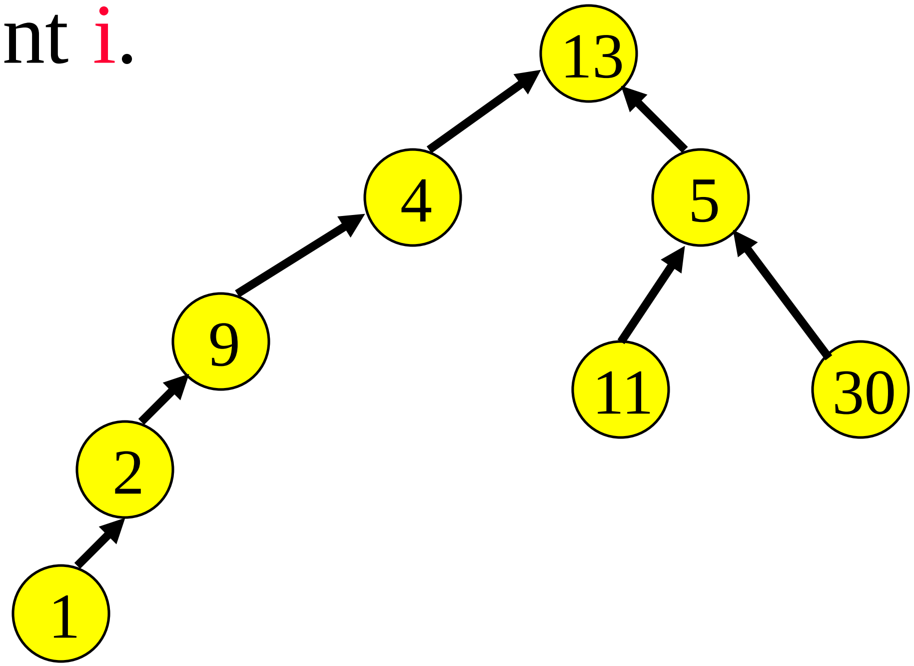
Example



(Only some table entries are shown.)

Better Representation

- Use an integer array parent[] such that parent[i] is the element that is the parent of element i.



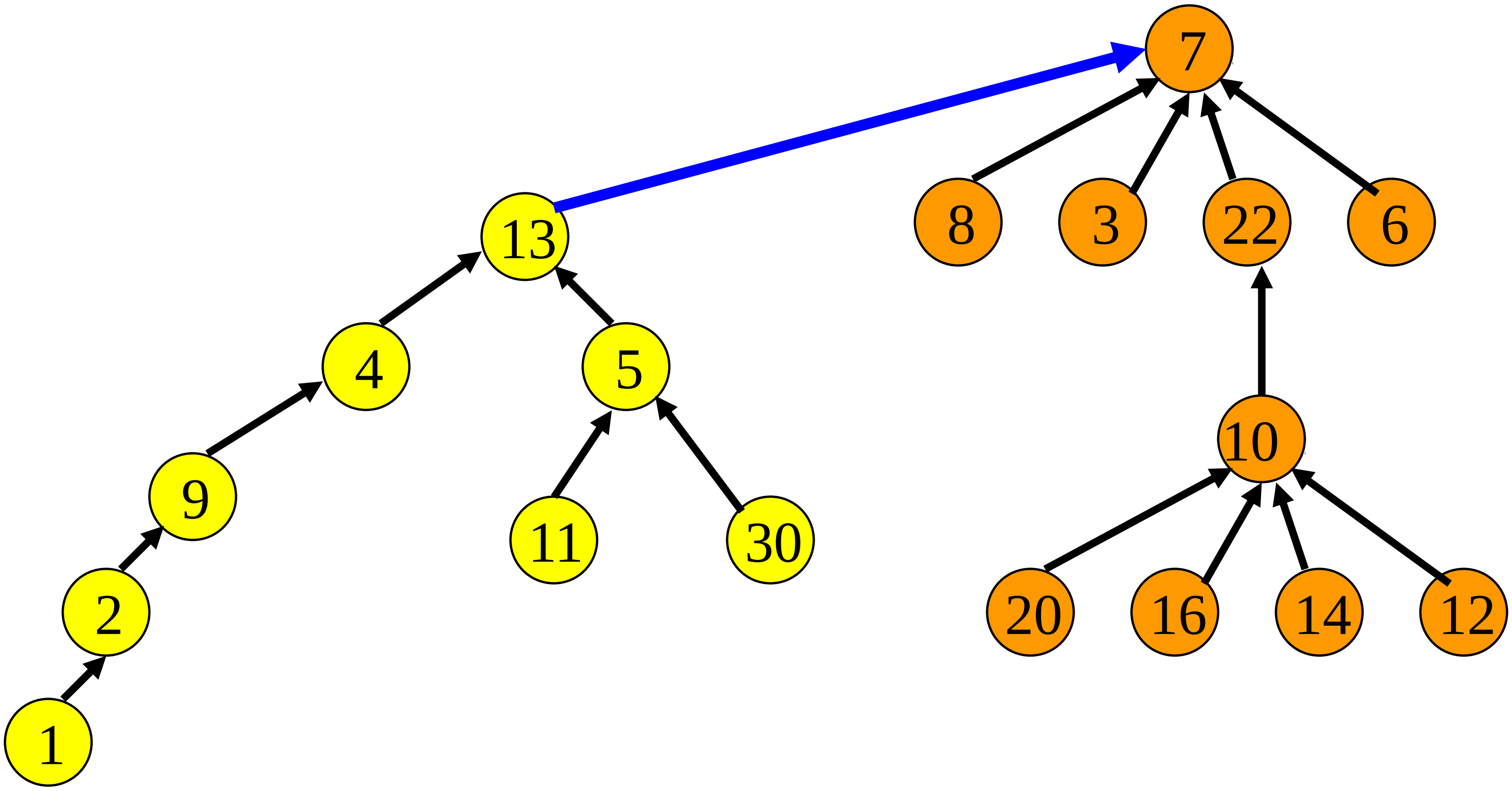
parent[]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	2	9		13	13				4		5		0		

Union Operation

- union(i,j)
  - i and j are the roots of two different trees, i != j.
- To unite the trees, make one tree a subtree of the other.
  - parent[j] = i

Union Example

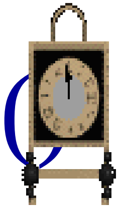


- union(7,13)

The Union Function

```
void simpleUnion(int i, int j)
    {parent[i] = j;}
```

Time Complexity Of simpleUnion()



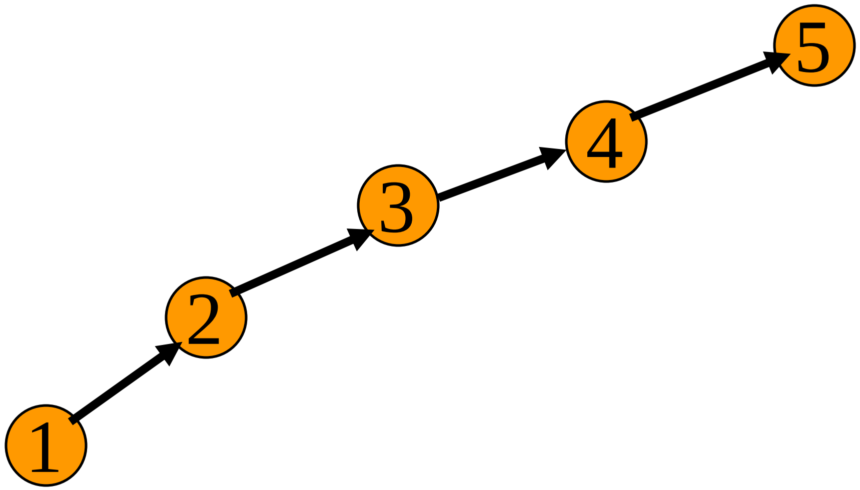
- O(1)

The Find Function

```
int simpleFind(int i)
{
    while (parent[i] >= 0)
      i = parent[i];  // move up the tree
    return i;
}
```

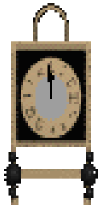
Time Complexity of simpleFind()

- Tree height may equal number of elements in tree.
  - union(2,1), union(3,2), union(4,3), union(5,4)…

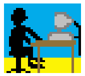


So complexity is O(u).

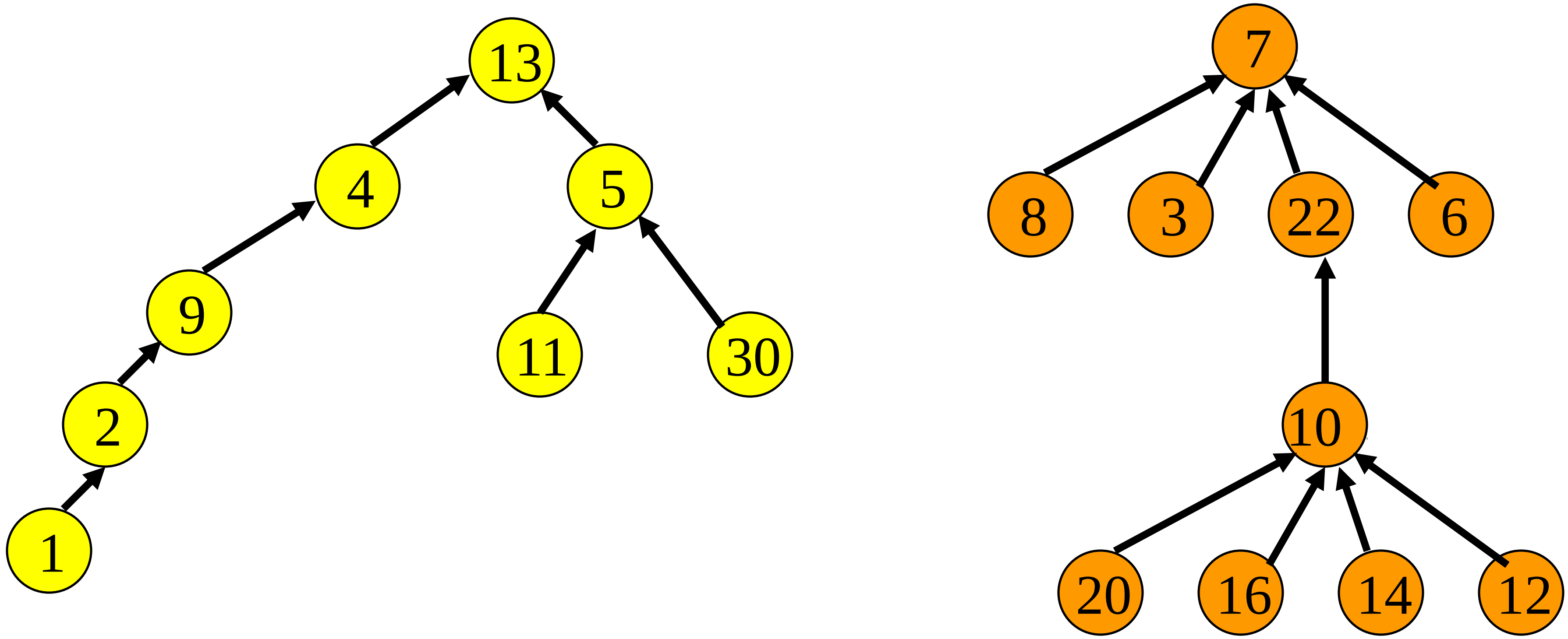
u Unions and f Find Operations



- $O(u + uf) = O(uf)$
- Time to initialize $parent[i] = 0$ for all i is $O(n)$.
- Total time is $O(n + uf)$.
- Worse than using a chain!
- Back to the drawing board.



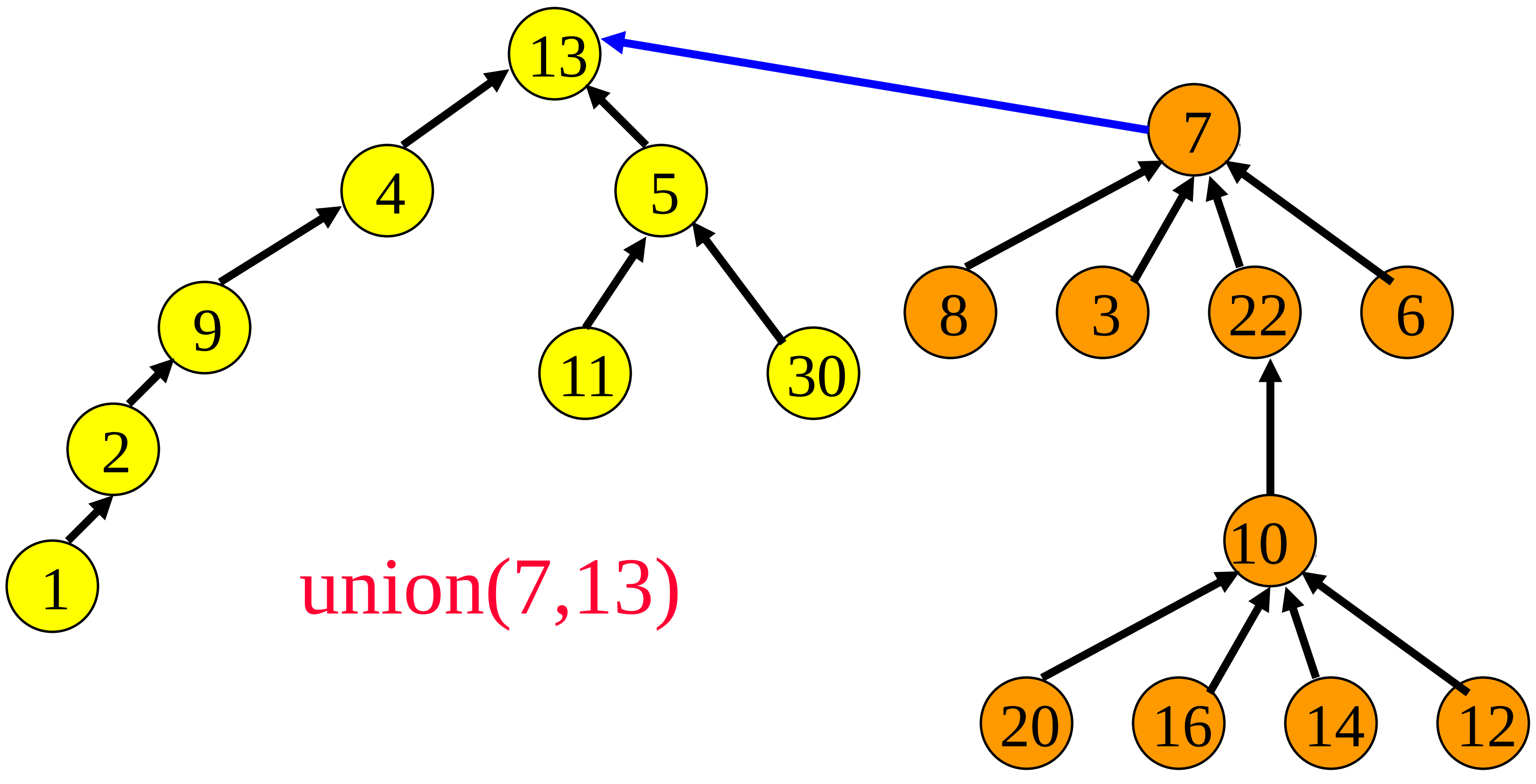
Smart Union Strategies



- union(7,13)
- Which tree should become a subtree of the other?

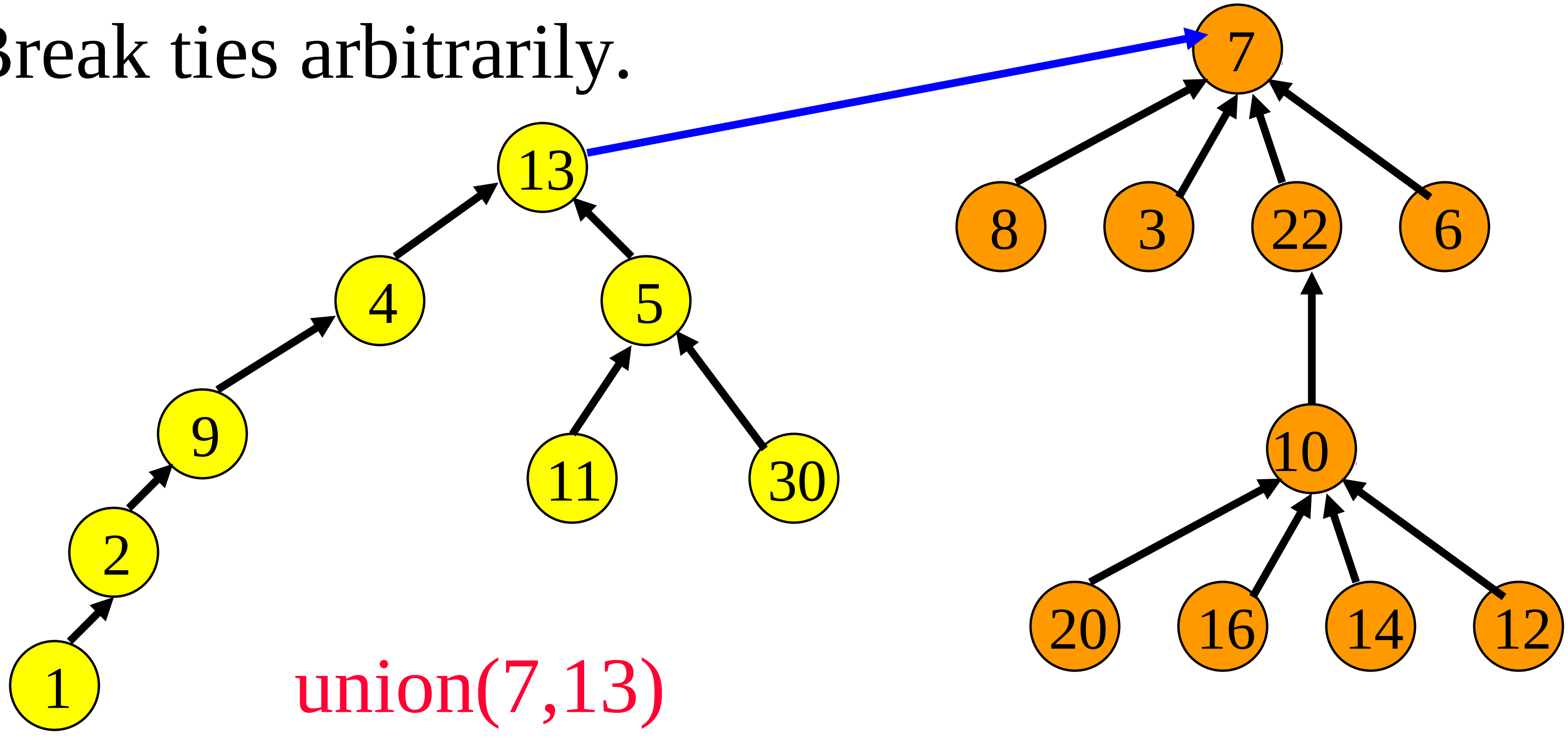
Height Rule

- Make tree with smaller height a subtree of the other tree.
- Break ties arbitrarily.



Weight Rule

- Make tree with fewer number of elements a subtree of the other tree.
- Break ties arbitrarily.



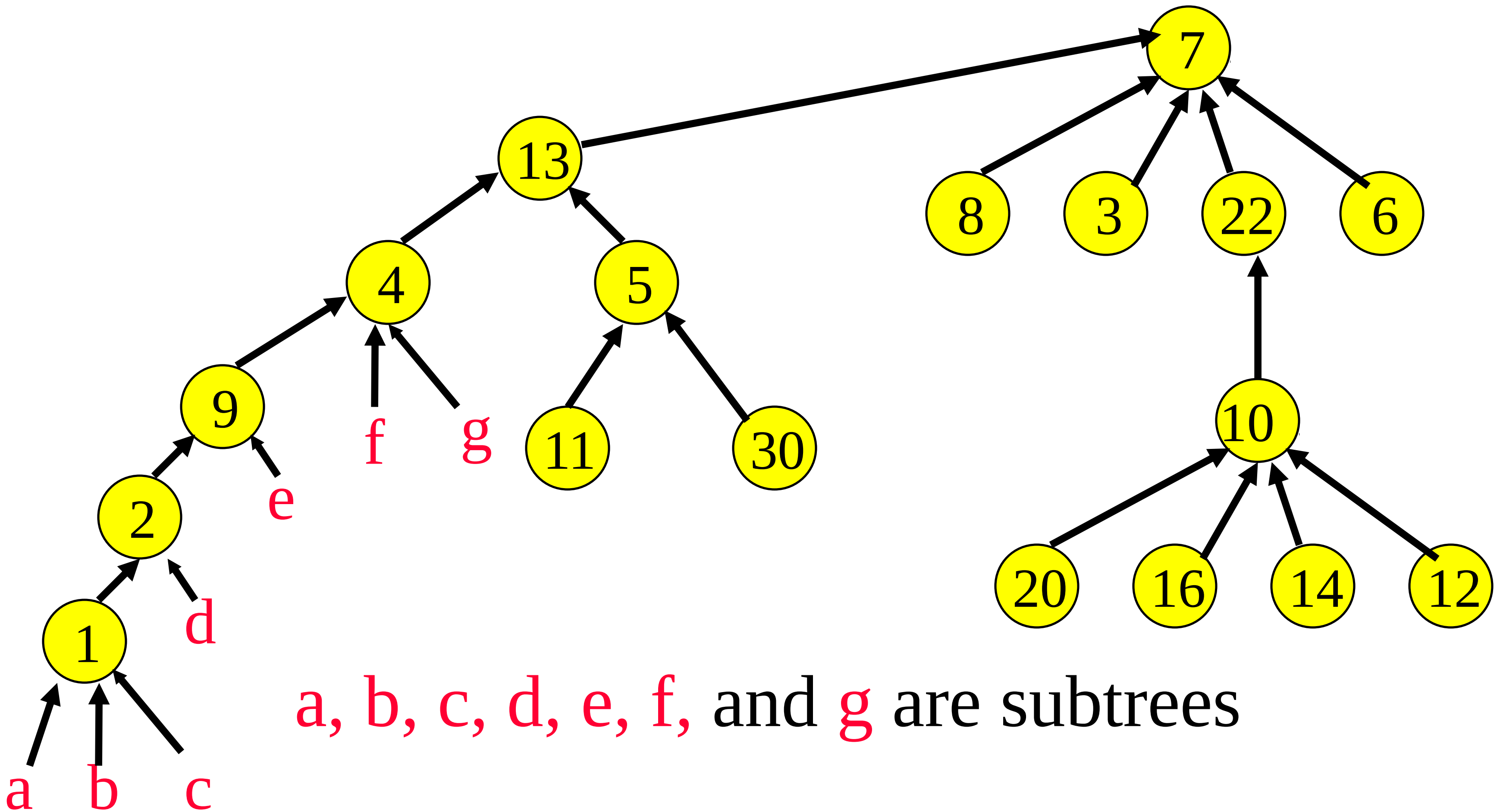
Implementation

- Root of each tree must record either its height or the number of elements in the tree.
- When a union is done using the height rule, the height increases only when two trees of equal height are united.
- When the weight rule is used, the weight of the new tree is the sum of the weights of the trees that are united.

Height Of A Tree

- Suppose we start with single element trees and perform unions using either the height or the weight rule.
- The height of a tree with p elements is at most $\text{floor}(\log_2 p) + 1$.
- Proof is by induction on p. See text.

Sprucing Up The Find Method

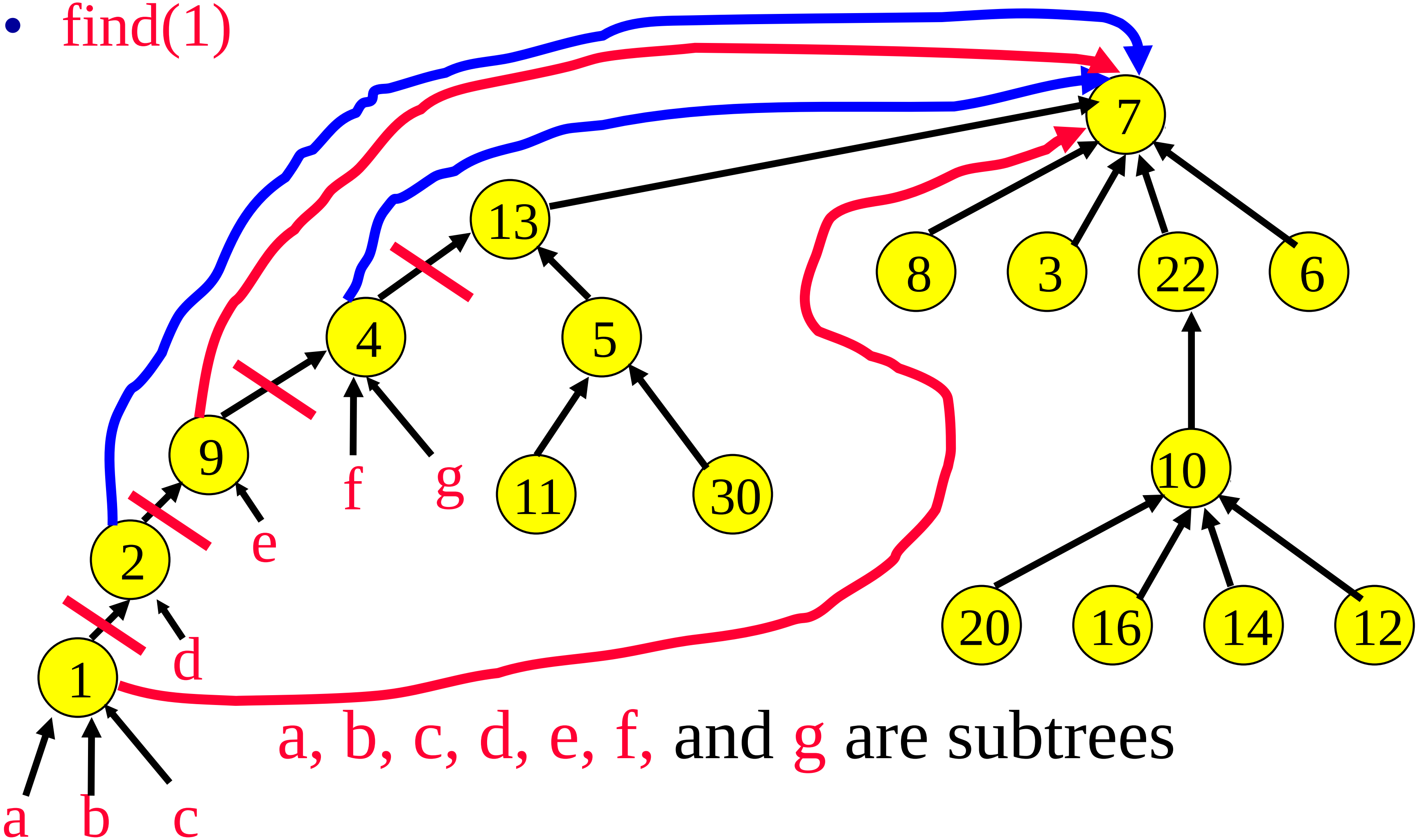


a, b, c, d, e, f, and g are subtrees

- find(1)
- Do additional work to make future finds easier.

Path Compaction

- Make all nodes on find path point to tree root.
- find(1)

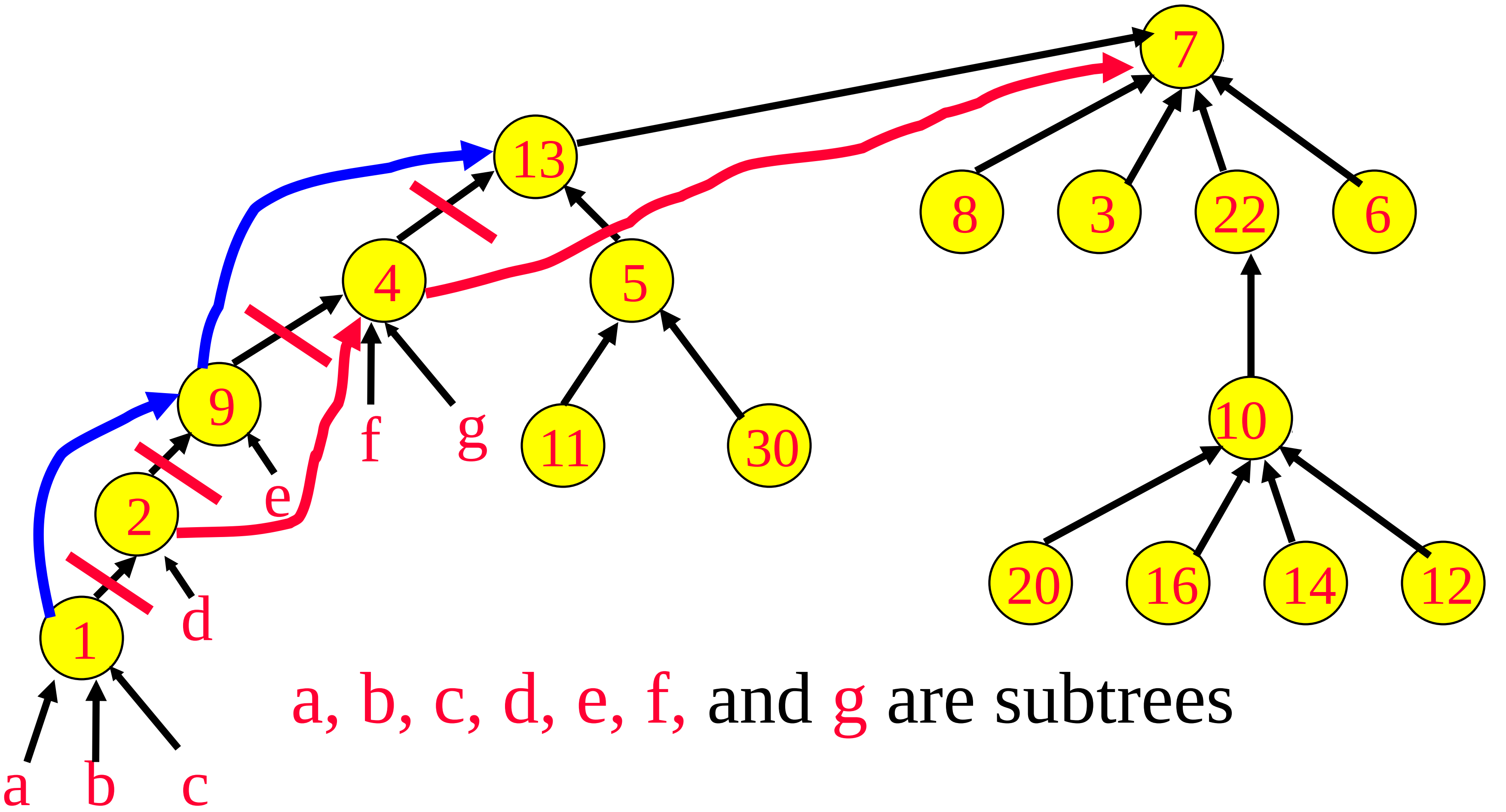


a, b, c, d, e, f, and g are subtrees

Makes two passes up the tree.

Path Splitting

- Nodes on find path point to former grandparent.
- find(1)

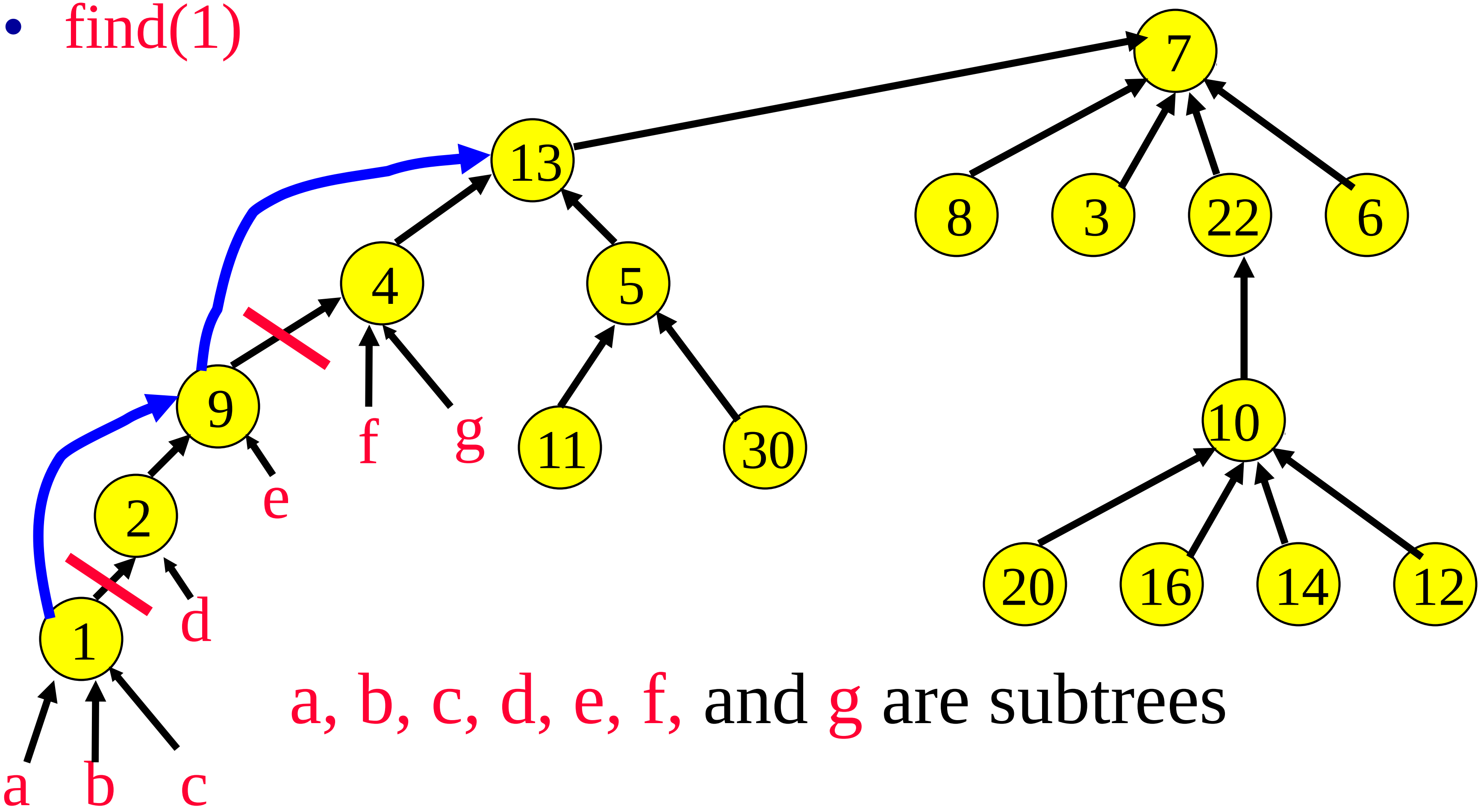


a, b, c, d, e, f, and g are subtrees

Makes only one pass up the tree.

Path Halving

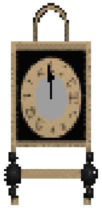
- Parent pointer in every other node on find path is changed to former grandparent.
- find(1)



a, b, c, d, e, f, and g are subtrees

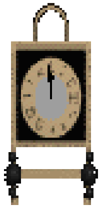
Changes half as many pointers.

Time Complexity



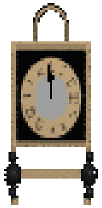
- Ackermann's function.
  - $A(i,j) = 2^j$, $i = 1$ and $j >= 1$
  - $A(i,j) = A(i-1,2)$, $i >= 2$ and $j = 1$
  - $A(i,j) = A(i-1,A(i,j-1))$, $i, j >= 2$
- Inverse of Ackermann's function.
  - $\alpha(p,q) = \min\{z>=1 \mid A(z, p/q) > \log_2 q\}$, $p >= q >= 1$

Time Complexity



- Ackermann's function grows very rapidly as i and j are increased.
  - $A(2,4) = 2^{65,536}$
- The inverse function grows very slowly.
  - $\alpha(p,q) < 5$ until $q = 2^{A(4,1)}$
  - $A(4,1) = A(2,16) >>>> A(2,4)$
- In the analysis of the union-find problem, q is the number, n, of elements; $p = n + f$; and $u >= n/2$.
- For all practical purposes, $\alpha(p,q) < 5$.

Time Complexity



Lemma 5.6 [Tarjan and Van Leeuwen]

Let $T(f,u)$ be the maximum time required to process any intermixed sequence of f finds and u unions. Assume that $u >= n/2$.

$$k_1*(n + f*\alpha(f+n, n)) <= T(f,u) <= k_2*(n + f*\alpha(f+n, n))$$

where k_1 and k_2 are constants.

These bounds apply when we start with singleton sets and use either the weight or height rule for unions and any one of the path compression methods for a find.