CS 6212 DESIGN AND ANALYSIS OF ALGORITHMS

LECTURE: GRAPH TRAVERSAL TECHNIQUES

Instructor: Abdou Youssef

OBJECTIVES OF THIS LECTURE

By the end of this lecture, you will be able to:

- Describe and apply two graph traversal techniques, namely, Depth-First Search (DFS) and Breadth-First Search (BFS), as well as three tree traversal techniques. Namely, inorder, preoder and postorder traversal
- Apply DFS and BFS to solve some standard graph problems, including connectivity, and special cases of shortest paths and minimum spanning trees
- Apply DFS to check for biconnectivity and identify articulation points
- Use the concepts and insights gained in this lecture to develop new graph algorithms for some interesting problems

OUTLINE

By the end of this lecture, you will be able to:

- Definition of graph traversal
- Tree traversal techniques and some quick applications
- Depth-first search: technique, implementation, and applications
- Breadth-first search: technique, implementation, and applications
- Biconnectivity application of depth-first search

INTRODUCTION

- A graph search (or traversal) technique visits all the nodes in an input graph in a systematic fashion
  - The nodes are visited in some particular order
  - No node is visited multiple times (but it can be crossed multiple times), and no node is left out
- Two standard graph search techniques have been widely used:
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)
- In the case of rooted binary trees, 3 traversal techniques are widely used:

  (1) Inorder Traversal (2) Preorder Traversal (3) Postorder Traversal
- The tree traversal techniques will be reviewed very briefly
- BFS and DFS will be covered in detail, and applications given

TREE TRAVERSAL TECHNIQUES

```
Procedure inorder(input:T)
begin
    if T = null then:  return; endif

    // traverse the left subtree
    // recursively
    inorder(T.left);

    // visit/process the root
    visit(T);

    // traverse the right subtree
    // recursively
    inorder(T.right);
end
// Also called LNR traversal
```

Time: $T(n) = O(n)$
because every node in the tree is processed/crossed just once

```
Procedure preorder(input:T)
begin
    if T = null then:  return; endif

    // visit/process the root
    visit(T);

    // traverse the left subtree
    // recursively
    preorder(T.left);

    // traverse the right subtree
    // recursively
    preorder(T.right);
end
// Also called NLR traversal
```

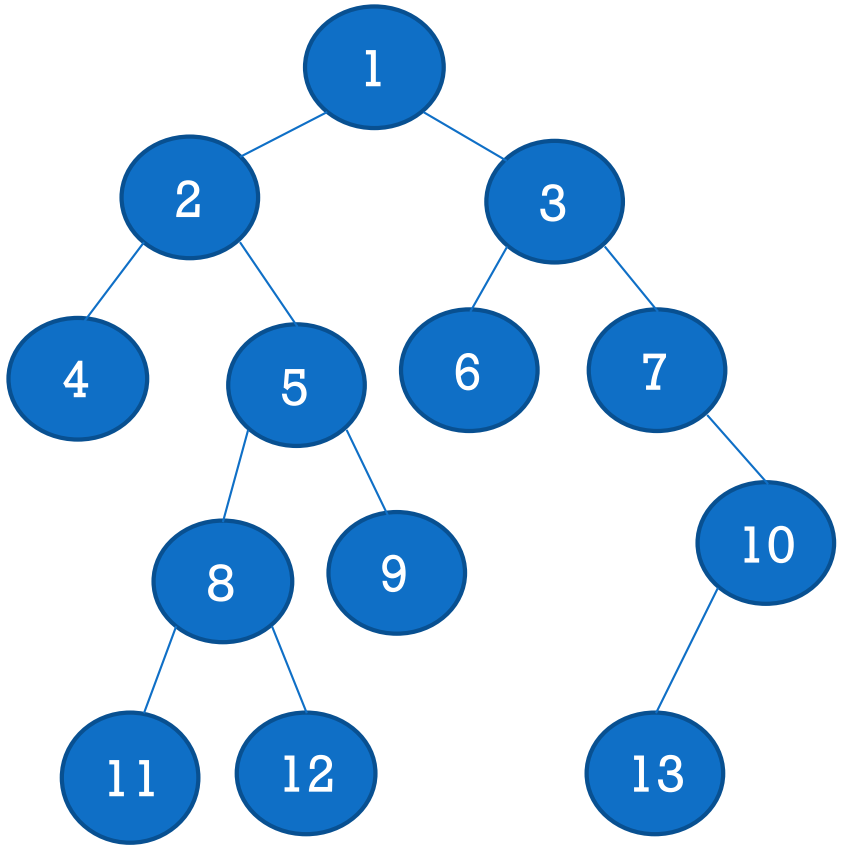
Time: $T(n) = O(n)$
because every node in the tree is processed/crossed just once

```
Procedure postorder (input:T)
begin
    if T = null then:  return; endif

    // traverse the left subtree
    // recursively
    postorder(T.left);

    // traverse the right subtree
    // recursively
    postorder(T.right);

    // visit/process the root
    visit(T);
end
// Also called LRN traversal
```

Time: $T(n) = O(n)$
because every node in the tree is processed/crossed just once

ILLUSTRATION OF THE THREE TREE TRAVERSAL TECHNIQUES

- Inorder: 4, 2, 11, 8, 12, 5, 9, 1, 6, 3, 7, 13 10
- Preorder: 1, 2, 4, 5, 8, 11, 12, 9, 3, 6, 7, 10, 13
- Postorder: 4, 11, 12, 8, 9, 8, 2, 6, 13, 10, 7, 3, 1



APPLICATIONS OF TREE TRAVERSAL TECHNIQUES

- Compiling (and evaluating) arithmetic expressions:
  - Infix notation: inorder traversal
  - Prefix notation: preorder traversal
  - Postfix notation: postorder traversal
- We will not say more about the compiler applications, but you can read about them by following the links if you wish
- Sorting a Binary Search Tree
  - If apply inorder traversal on a BST, the data of the tree get sorted
  - Thus, sorting a BST takes O(n) time

LESSONS LEARNED SO FAR

- Tree traversal techniques are simple, recursive, and linear in time
- Sorting a BST is done by applying inorder traversal on it

EXERCISES

- Can two different binary trees yield the same inorder traversal sequence? The same preorder traversal sequence? The same postorder traversal sequence? Prove your answer
- Write a recursive algorithm that takes as input of the postorder traversal sequence and the in-order traversal sequence of a binary tree, and derives as output the actual binary tree

DEPTH-FIRST SEARCH (DFS)

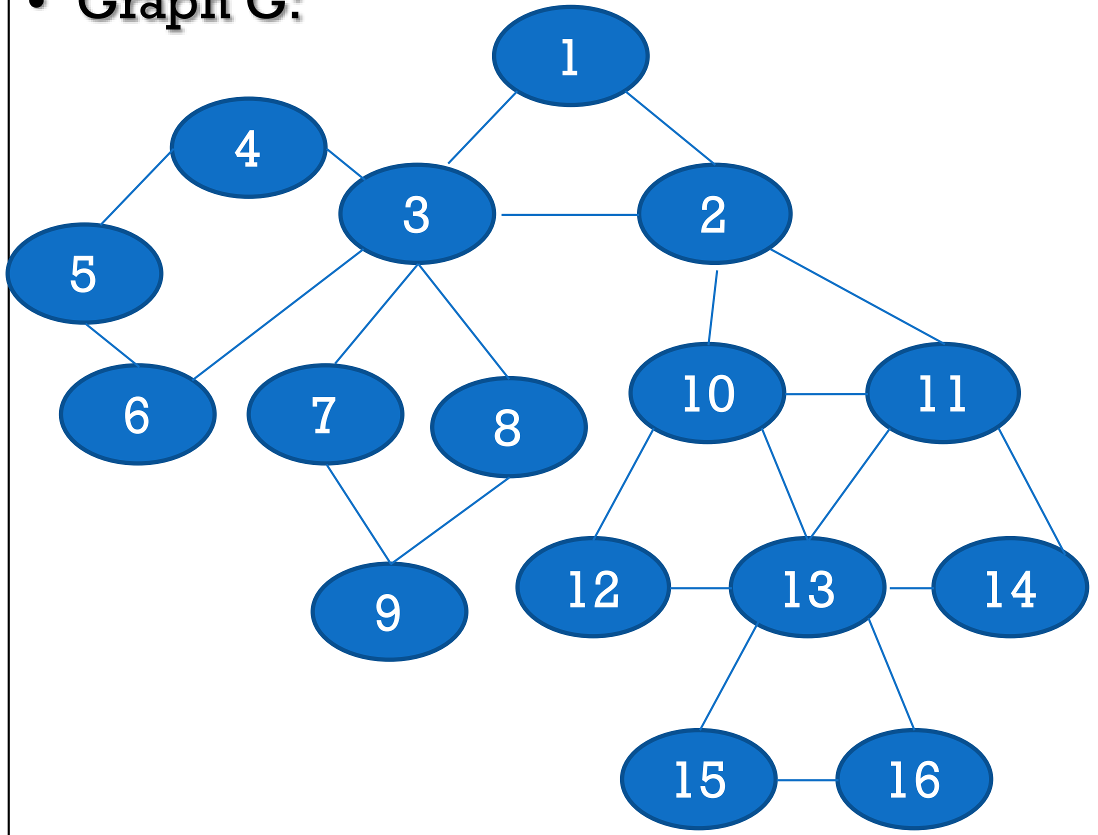
DFS follows these steps/rules:

Better to illustrate on an example, next

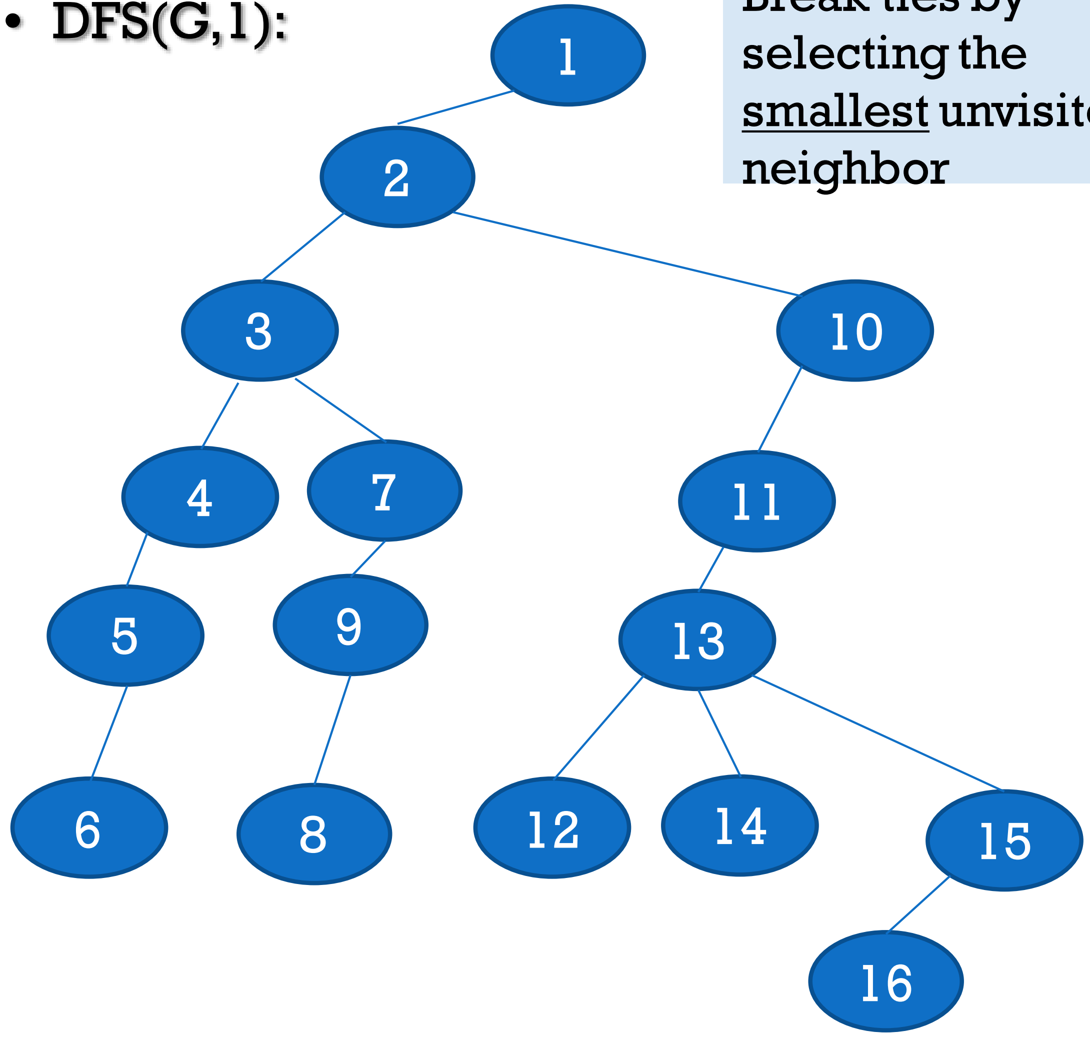
1. Select an unvisited node s, visit it, and treat as the current node
2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
3. If the current node has no unvisited neighbors, backtrack to its parent, and make that the new current node;
4. Repeat the above two steps until no more nodes can be visited;
5. If there are still unvisited nodes, repeat from step 1;

DFS ILLUSTRATION

- Graph G:



- DFS(G,1):



Break ties by selecting the <u>smallest</u> unvisited neighbor

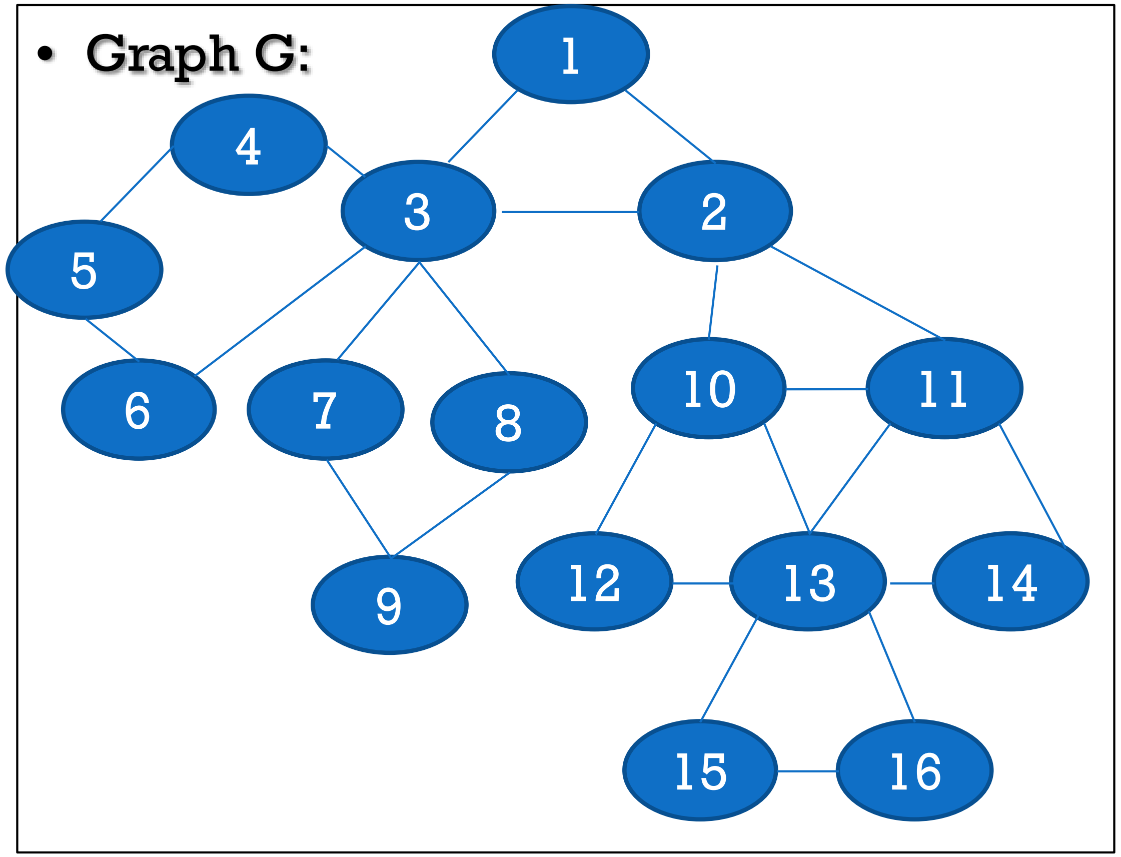
DFS IMPLEMENTATION
-- WHY --

- Remark: whenever we got stuck, we backtracked to where we came from. How?
  - Using our human eyes!
  - Algorithms don't have eyes (usually)
- Observe that the last node you came from is the first node you go back to
- This suggests a **stack** to remember the order in which to backtrack to nodes

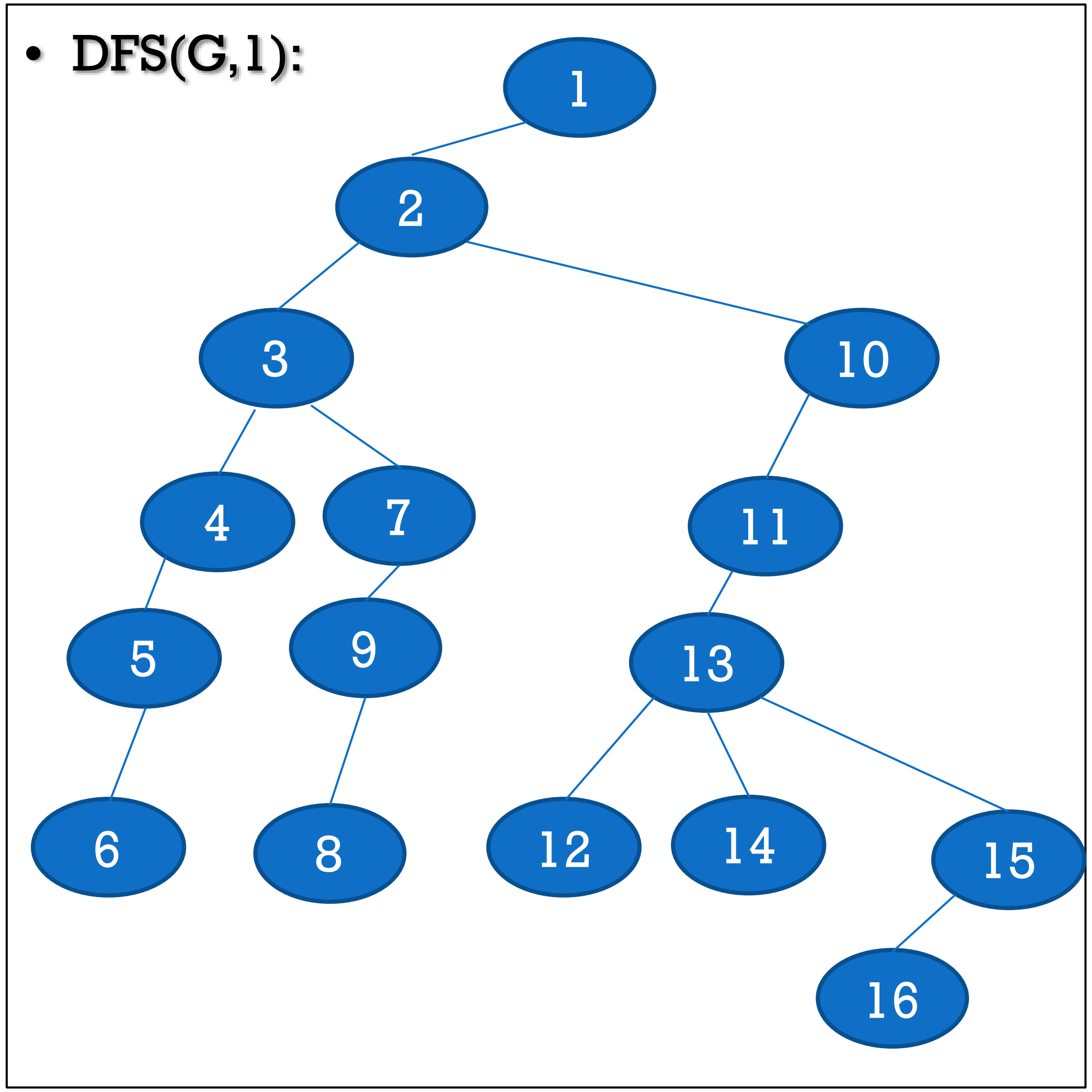
DFS IMPLEMENTATION

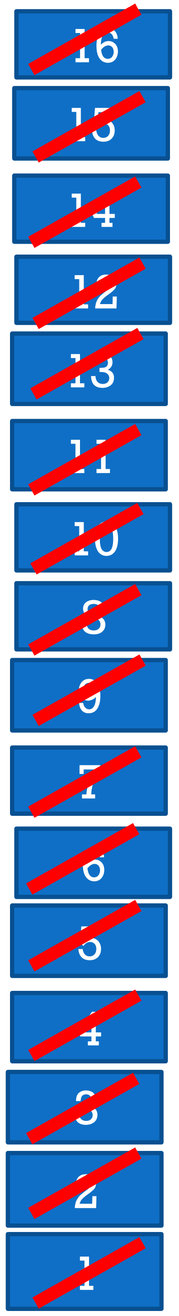
-- ILLUSTRATION: USING STACKS --

- Graph G:



- DFS(G,1):





- When you visit a node: push it onto stack
- The node to backtrack to: top of stack
- When you backtrack from a node: pop it
- When stack is empty, current DFS round is done

DFS IMPLEMENTATION
-- CODE: USING STACKS--

```
Procedure DFS(input: graph G)
begin
    Stack S;
    int x, y, v;
    while (G has an unvisited node) do
        // pick one of them; break tie by picking min
        v := an unvisited node; // a starting node
        visit(v);    push(v,S);
        while (S is not empty) do
            x := top(S);  // current node
            if (x has an unvisited neighbor y) then
                visit(y); // break tie, e.g., pick min
                push(y,S);
            else
                pop(S); // backtrack to previous node
            endif
        endwhile
    endwhile
end
```

Time complexity:

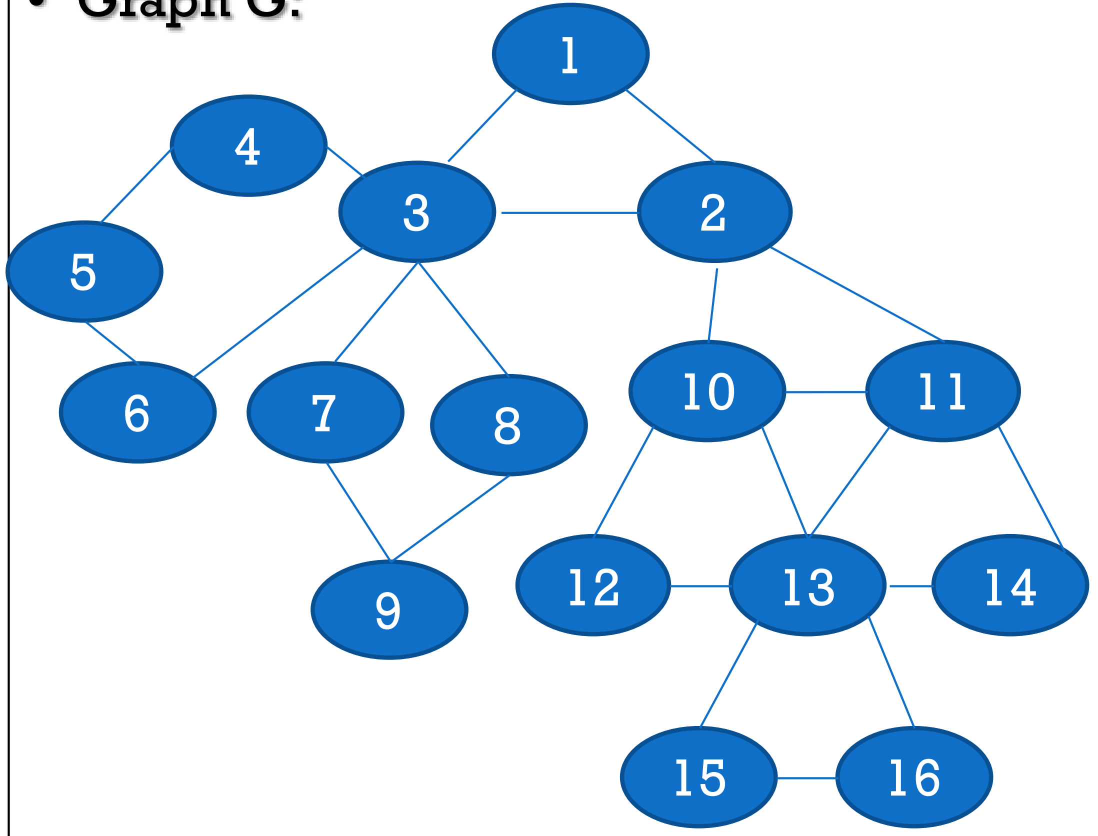
- Every node is visited once, but could be crossed multiple times (by backtracking to it multiple times).
- So the number of nodes is not a good indicator of time
- But ever edge in G is "traversed" at most twice:
  - One time to go from a node to a "child"
  - And another time to backtrack from a child to a parent
- Note that once you backtracked ***from*** a node, you never come back to it
- Therefore, the time is **O(|E|+n)**

OBSERVATIONS
-- DFS ON CONNECTED GRAPHS: DFS TREE --

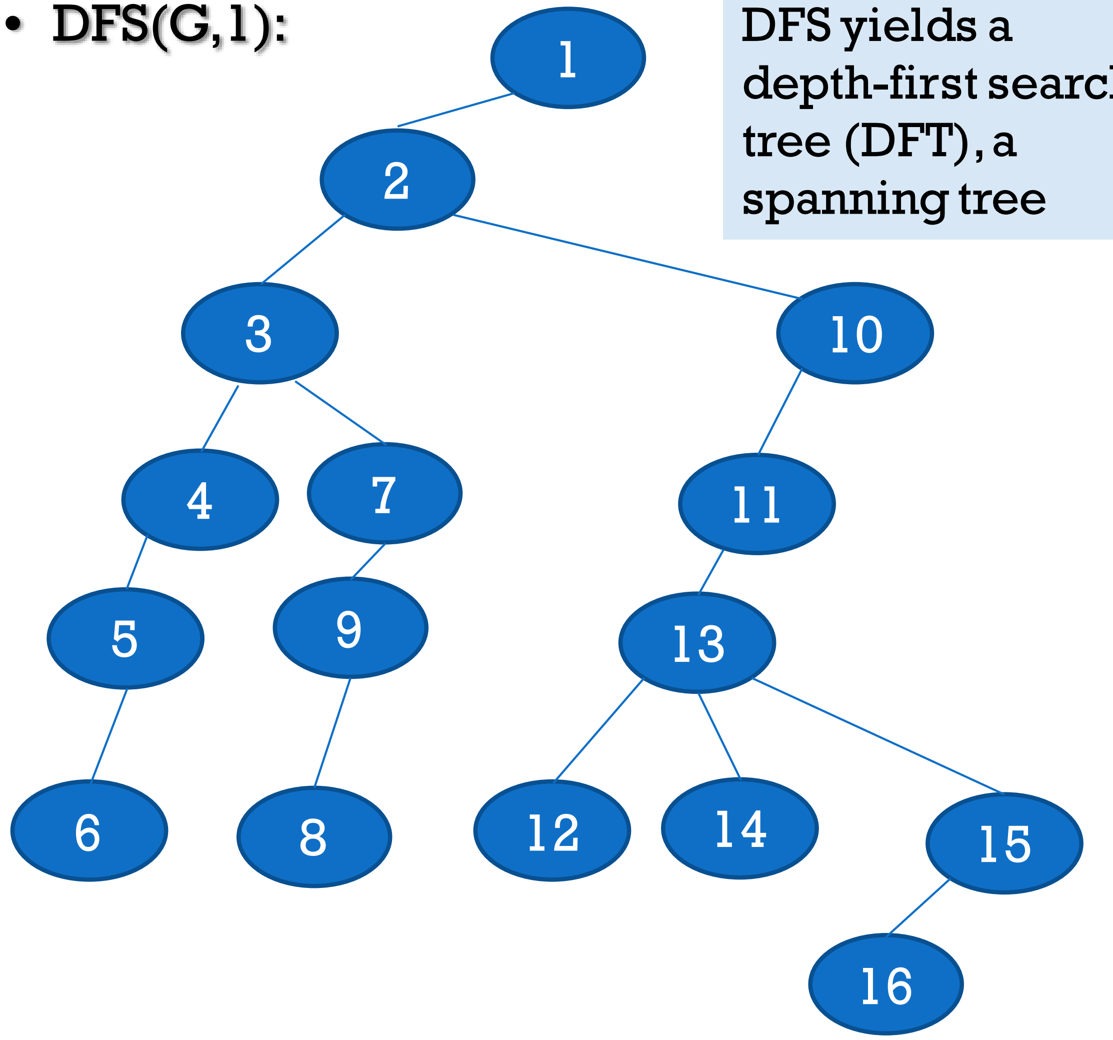
- When you do a DFS on a graph, and you draw a copy of a node when you visit it, and you draw an edge to a every new node you visit from your current node, <u>you end up with a tree</u>
- That tree is called a depth-first search tree (or simply DFT)
- If the graph G is connected (i.e., there is at least one path between every pair of nodes), then:
  - Doing a DFS on G from any arbitrary starting node will visit **<u>all</u>** the nodes in G
  - The DFT tree is a spanning tree of G

DFS ON CONNECTED GRAPHS
-- YIELDING A DFT --

- Graph G:



- DFS(G,1):

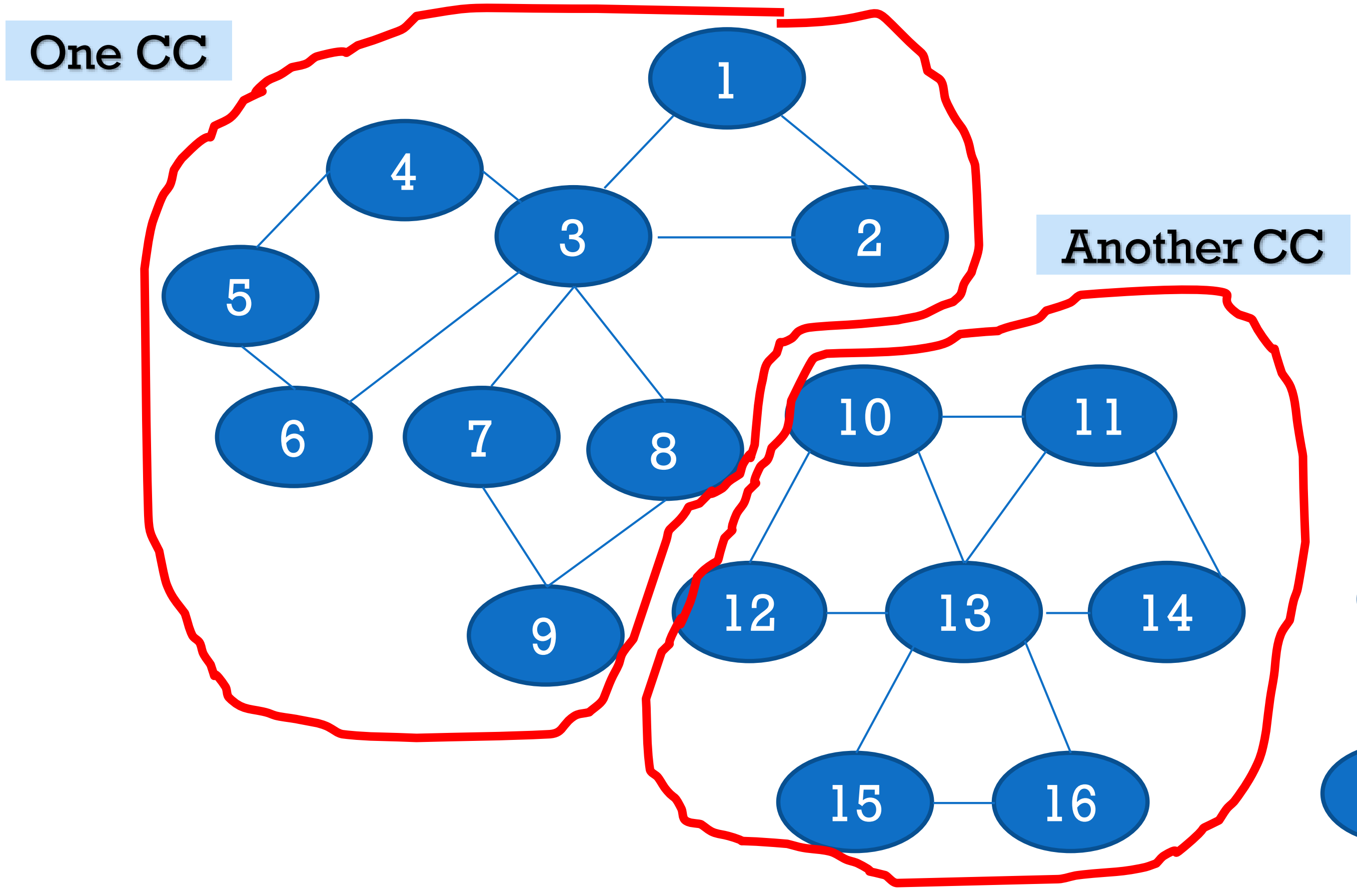


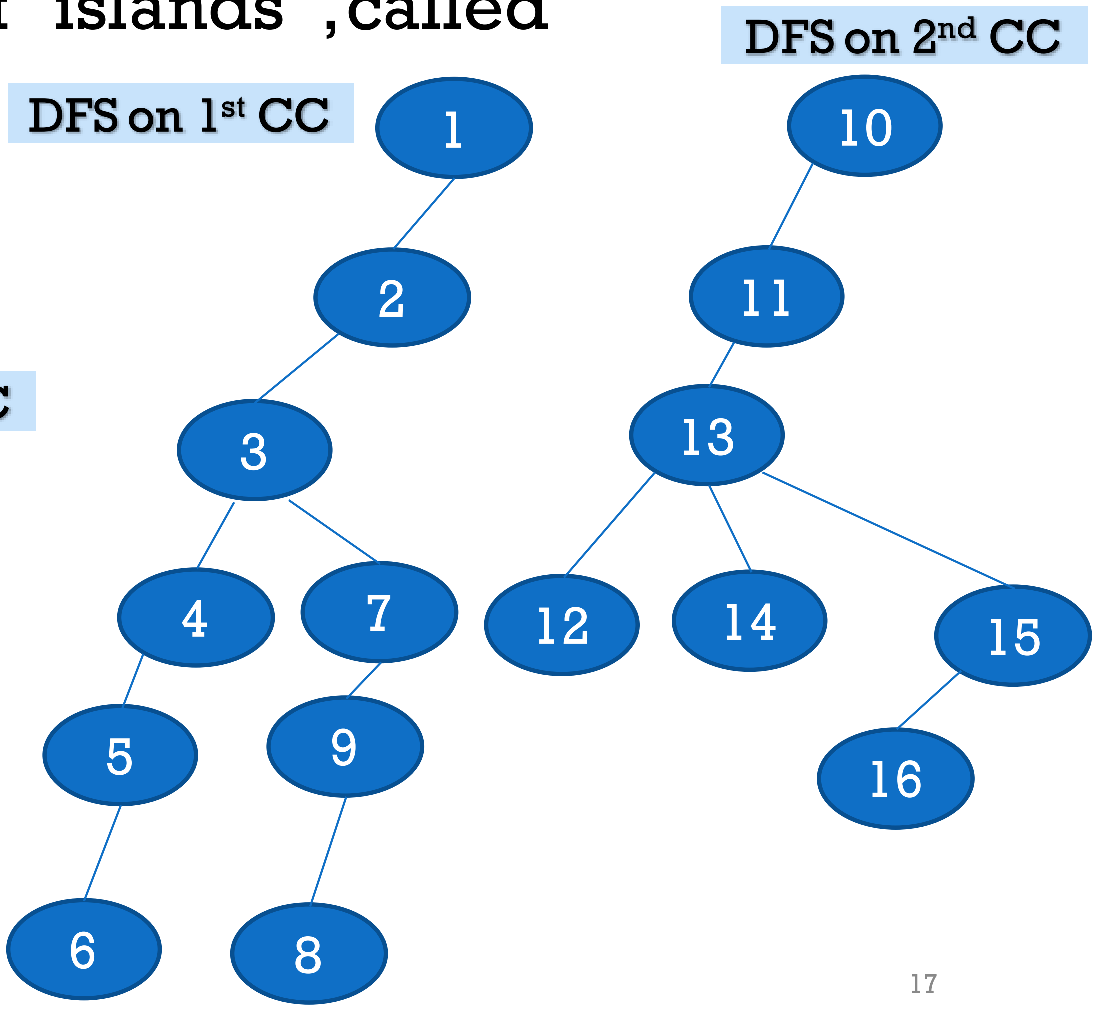
DFS yields a depth-first search tree (DFT), a spanning tree

OBSERVATIONS

-- DFS ON DISCONNECTED GRAPHS: DFS FOREST --

- If G is not connected, it is made up of "islands", called connected components (CC)





DFS APPLICATIONS
-- CONNECTIVITY --

```
Procedure DFS(input: graph G)
begin
    Stack S;
    int x, y, v;
    int count=0; // number of connected components
    while (G has an unvisited node) do
        v := an unvisited node; // a starting node
        visit(v);   push(v,S);
        while (S is not empty) do
            x := top(S); // current node
            if (x has an unvisited neighbor y) then
                visit(y);  push(y,S);
            else
                pop(S); // backtrack to previous node
            endif
        endwhile
        count++;
    endwhile
end
```

Iterates as many times as there are CCs in G

- This traverses one full connected component
- If we keep track of the nodes visited in this round, we'll have the entire CC

At the end, "count" will have the number of CC's in G

- If count==1, then G is connected
- If count> 1, then G is disconnected

WHY IS CONNECTIVITY CHECKING IMPORTANT

- Ability to communicate with everyone
  - Our world is full of networks: computer networks, communication networks, etc.
  - Nodes in such networks communicate with one another very often
  - If the network is connected, that means every node can communicate with other nodes
  - Otherwise, some nodes are disconnected from other nodes
- Many graph applications work on connected graphs/components
  - Therefore, in those applications, the first step is to check if the input graph is connected
  - If G is not connected, the connected components are found first, and the application is then run on each CC separately

OTHER APPLICATIONS OF DFS

- Checking if an input graph G is a tree. How?
- Identification of which edges can be deleted while keeping the network connected (e.g., for cost saving measures)
- Minimum Spanning Trees when all edges have the same weight
  - If all edge weights are equal to w, then all spanning trees are of weight $(n-1)w$, and so all of them are MSTs
  - The DFT is then a MST, and can be found in O(|E|+n) time, which can be much less in the greedy $O(|E|\log|E|)$ time
- Finding single points of failure, i.e., individual nodes/edges whose removal disconnect the graph (more on that later)

LESSONS LEARNED SO FAR

- Tree traversal techniques are simple, recursive, and linear in time
- Sorting a BST is done by applying inorder traversal on it
- Depth-first search is a generic graph traversal technique that takes linear time (i.e., O(|E|+|V|)), and is easily implemented using stacks
- DFS has many applications, especially in connectivity, and yields a faster algorithm for the MST problem when the edges have the same weight

EXERCISES

- Give a recursive version of the DFS algorithm, which does not explicitly use a stack

BREADTH-FIRST SEARCH (BFS)

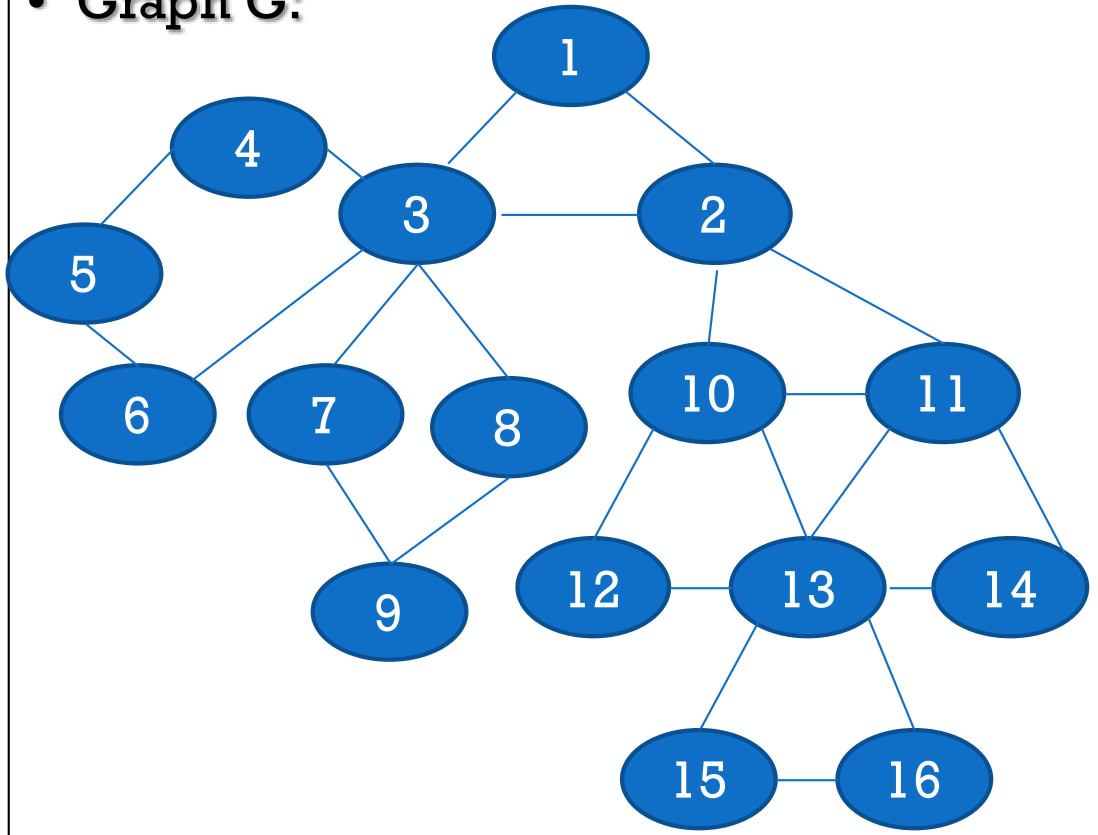
DFS follows these steps/rules:

Better to illustrate on an example, next

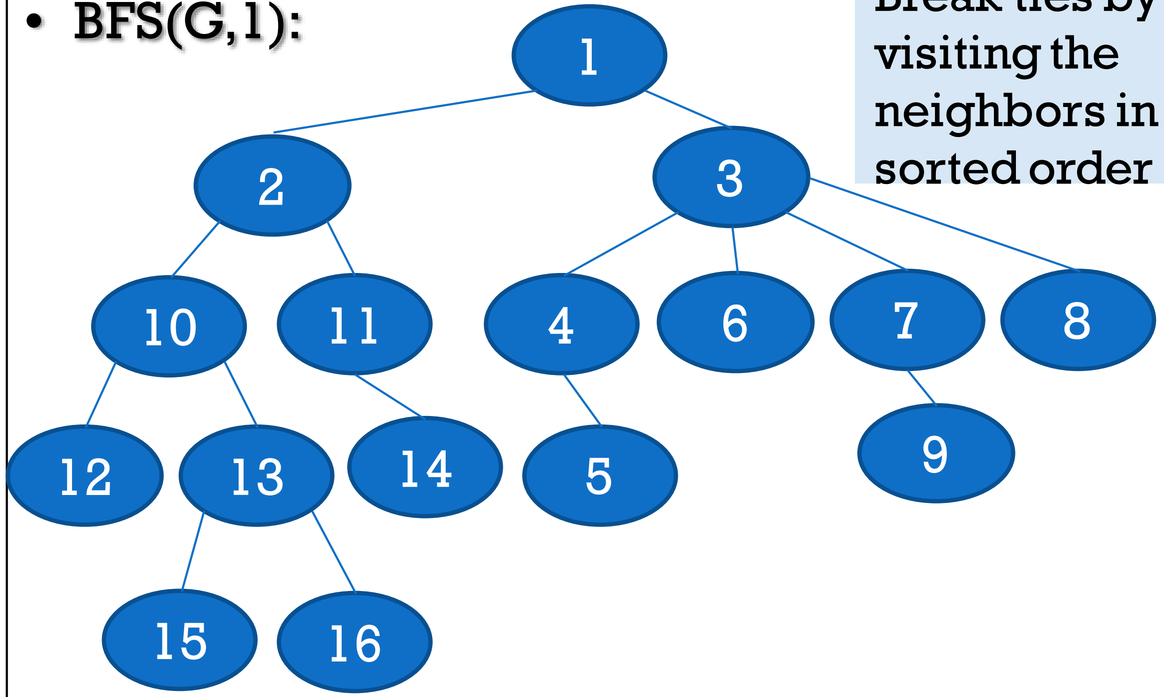
1. Select an unvisited node s, visit it, and treat as the current node (and the root of a BSF tree). Its level is called the current level.
2. From each node x in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of x. The newly visited nodes from this level form a new level that becomes the next current level.
3. Repeat the previous step until no more nodes can be visited.
4. If there are still unvisited nodes, repeat from Step 1.

BFS ILLUSTRATION

- Graph G:

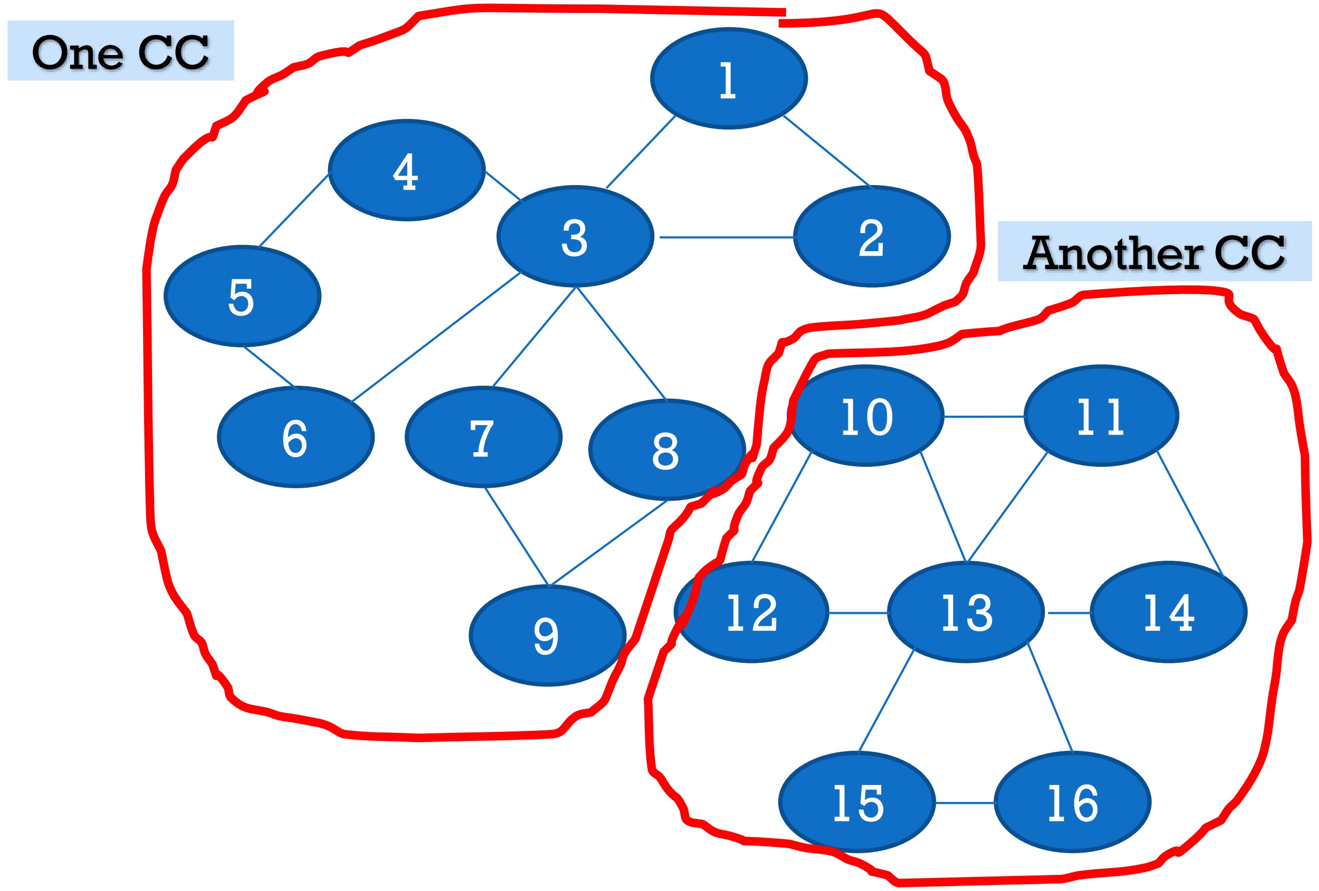


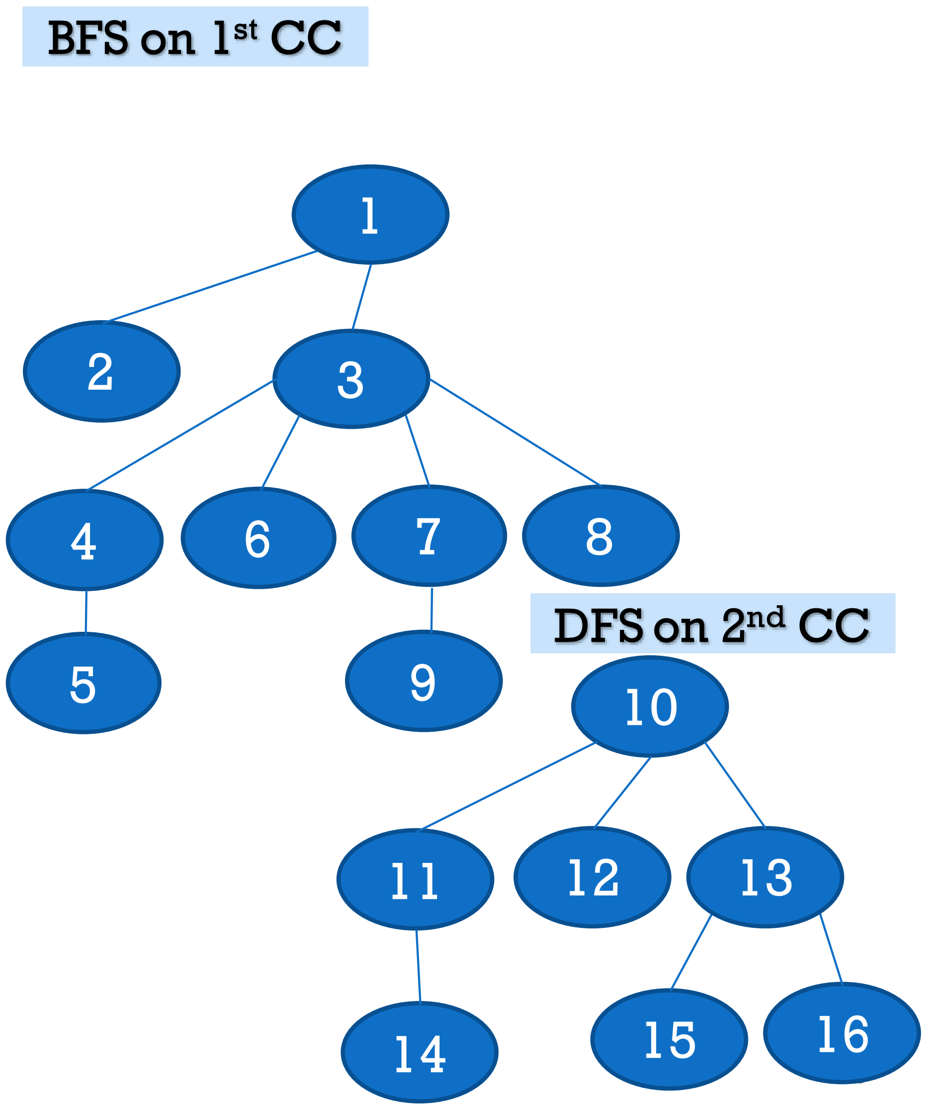
- BFS(G,1):



Break ties by visiting the neighbors in sorted order

BFS ON DISCONNECTED GRAPHS





OBSERVATIONS
-- BFS: BFS TREE OR FOREST--

- Like in DFS, when you do a BFS on a graph,
  - you get a tree if the graph is connected, called *Breadth-first search tree* (or simply BFT)
  - If the graph is disconnected, you get a forest (one tree per connected component), called *Breadth-first search forest*

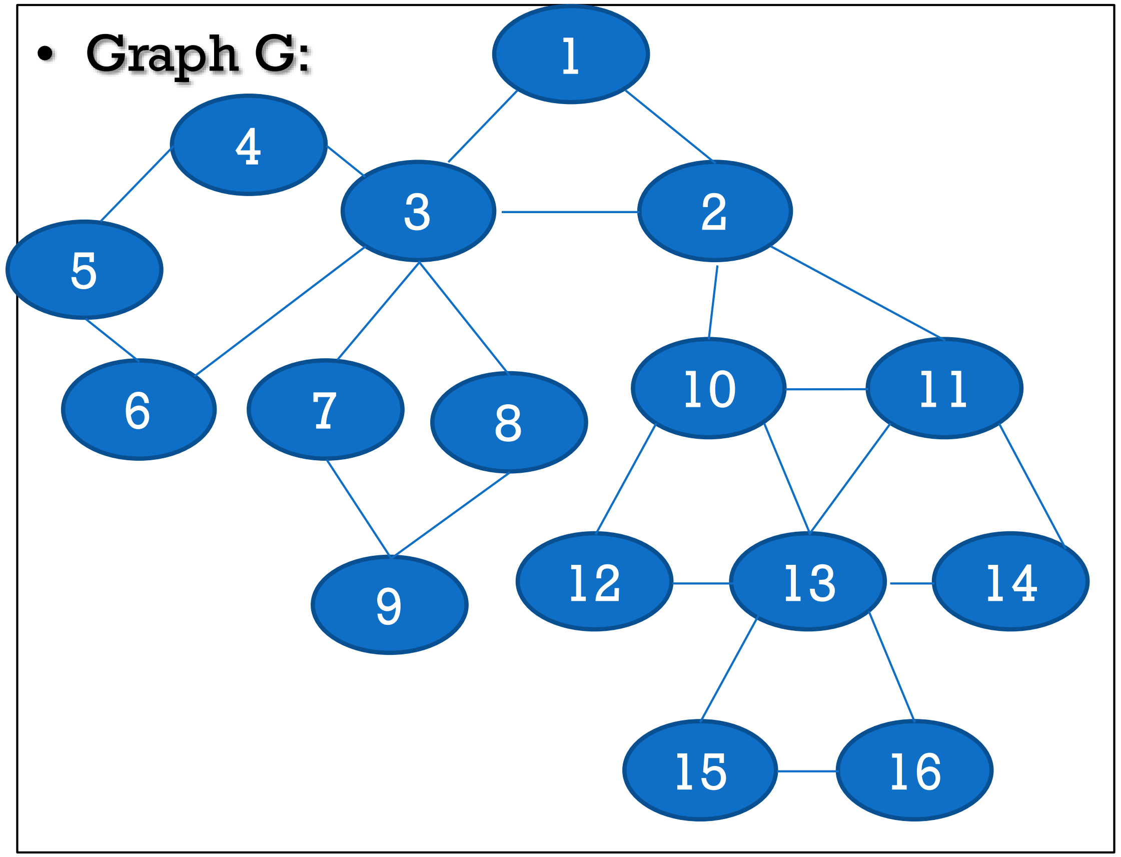
BFS IMPLEMENTATION
-- WHY --

Expand a node x: visit its unvisited neighbors

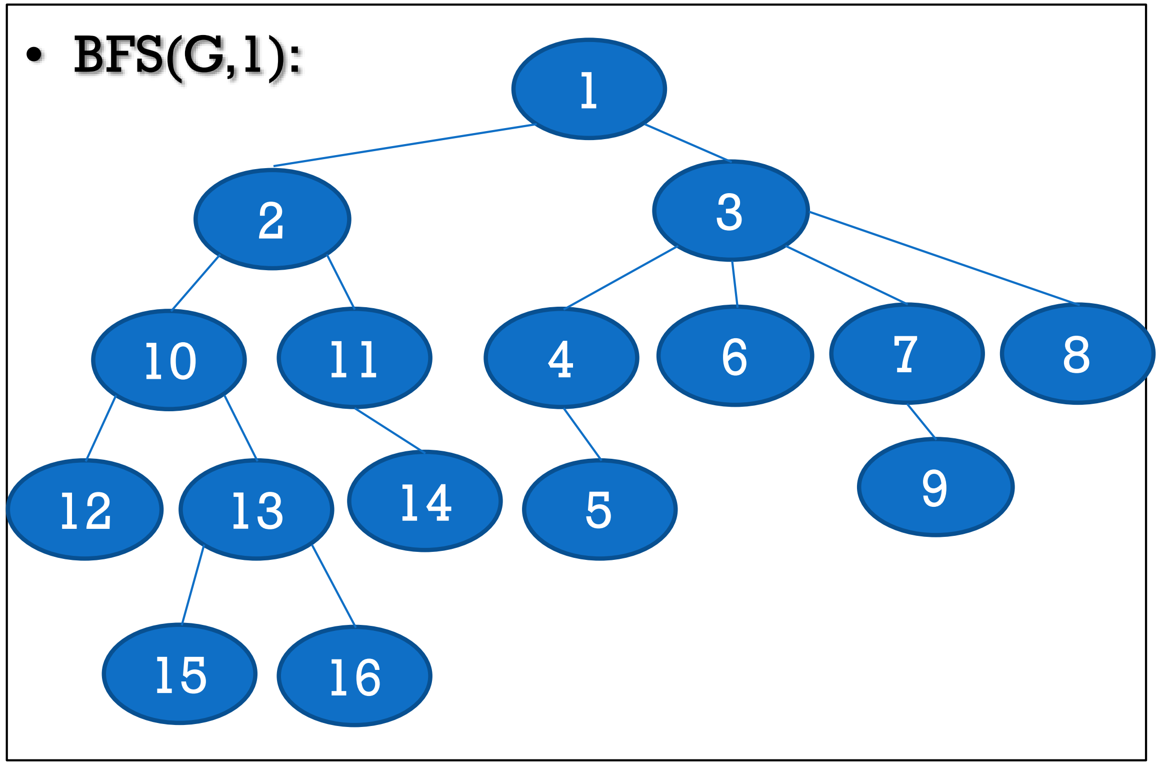
- Remarks:
  - The nodes in each "current" level are "expanded" from left to right, i.e., in the order in which they were initially visited: first-visit, first-expand
  - Whenever a level is completed, we moved to the next level
  - How did we follow the right order, and how did we find the next level?
  - Again, using our human eyes!
  - And again, algorithms don't have eyes (usually)
- The *first-visit, first-expand* observation above suggests a queue to remember the order in which to expand the (already visited) nodes

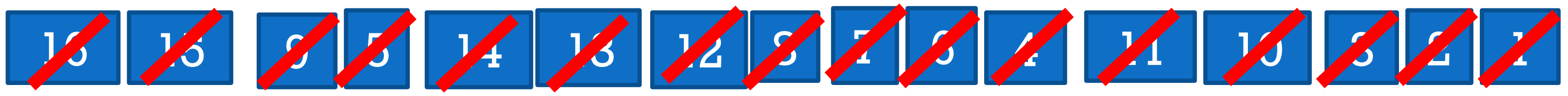
BFS IMPLEMENTATION
-- ILLUSTRATION: USING QUEUES--

- Graph G:



- BFS(G,1):





- When you visit a node, add it to the queue
- Next current node is head of the queue: get it and remove it from queue
- Stop when queue is empty

BFS IMPLEMENTATION -- CODE: USING QUEUES-

```
Procedure BFS(input: graph G)
begin
    Queue Q;
    int x, y, v;
    while (G has an unvisited node) do
        // pick one of them; break tie by picking min
        v := an unvisited node; // a starting node
        visit(v);     enqueuer(v,Q);
        while (Q is not empty) do
            x := dequeue(Q);  // current node
            for each unvisited neighbor y of do
                visit(y);// visit neighbors as sorted
                enqueue(y,W);
            endfor
        endwhile
    endwhile
end
```

Time complexity:

- Every node is visited once, but could be "touched" multiple times (to check if visited).
- So the number of nodes is not a good indicator of time
- But ever edge (x,y) in G is "traversed" twice:
    - Once to visit y from x
    - Another time from y to see if x has been visited
- Once x is expanded, edge from x to y is never crossed, b/c x is "left behind"
- Also once y is expanded, edge from y to x is never crossed
- Therefore, the time is **O(|E|+n)**

DFS APPLICATIONS -- CONNECTIVITY (MUCH LIKE DFS) --

```
Procedure BFS(input: graph G)
begin
    Queue Q;
    int x, y, v;
    int count=0; // number of connected components
    while (G has an unvisited node) do
        // pick one of them; break tie by picking min
        v := an unvisited node; // a starting node
        visit(v);      enqueuer(v,Q);
        while (Q is not empty) do
            x := dequeue(Q);  // current node
            for each unvisited neighbor y of do
                visit(y);// visit neighbors as sorted
                enqueue(y,Q);
            endfor
        endwhile
        count++;
    endwhile
end
```

Iterates as many times as there are CCs in G

- This traverses one full connected component
- If we keep track of the nodes visited in this round, we'll have the entire CC

At the end, "count" will have the number of CC's in G

- If count==1, then G is connected
- If count> 1, then G is disconnected

OTHER APPLICATIONS OF BFS

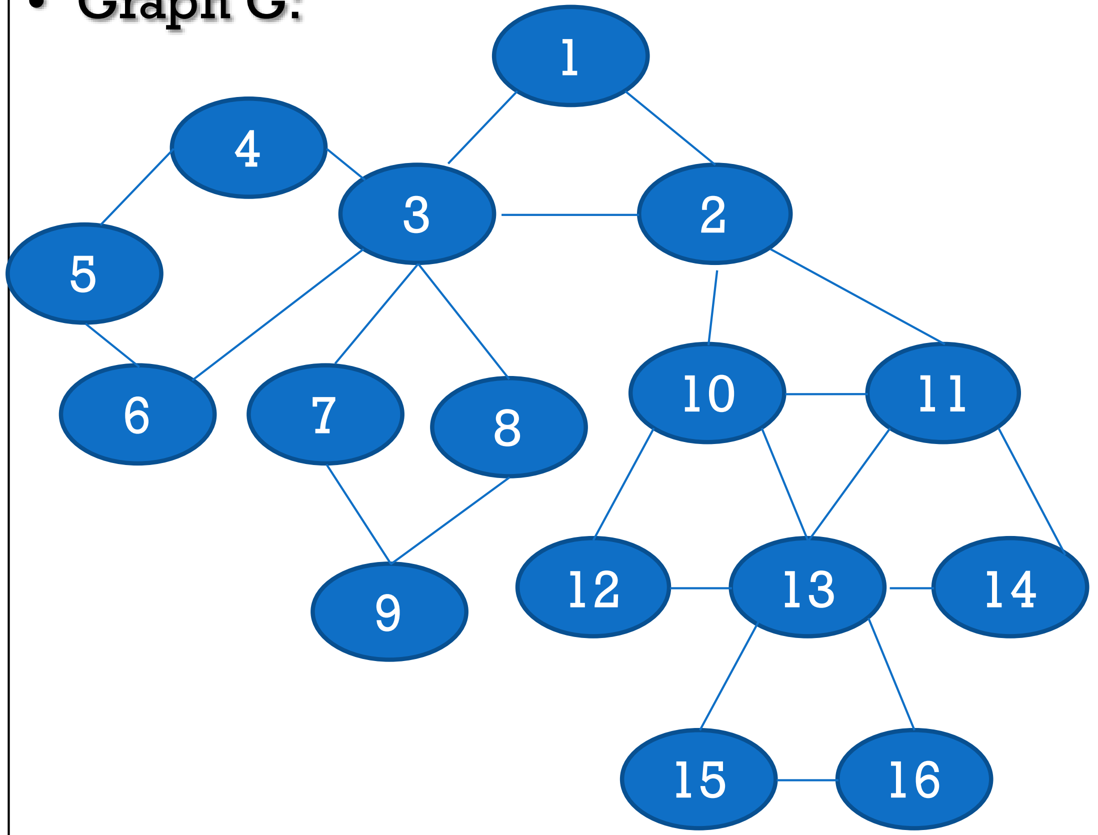
- Checking if an input graph G is a tree. How?
- Identification of which edges can be deleted while keeping the network connected (e.g., for cost saving measures)
- MSTs when all edges have the same weight: a BFT is a MST, and can be found in O(|E|+n) time, which is < the greedy $O(|E|\log|E|)$
- Reducing bandwidths in sparse matrices, etc. (beyond our scope)

OTHER APPLICATIONS OF BFS
-- SHORTEST PATHS WHEN ALL EDGES HAVE SAME WEIGHT (1) --

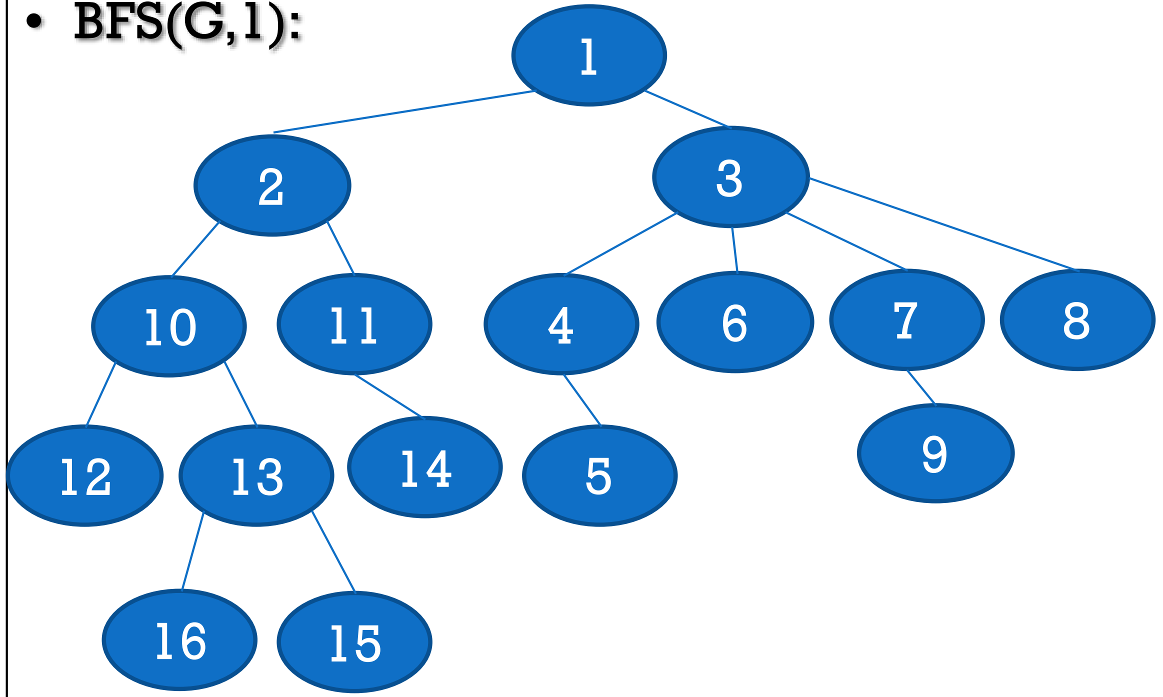
- Assume all the edges have the same weight (say 1) in a graph G
- **Theorem**: The BFS paths from the root to all the nodes are shortest paths
- **Proof**:
  - Label the levels of the BFT 0, 1, 2, …, top to bottom.
  - Prove by induction the level l that all nodes in level l are of distance l from the root, and that all nodes of distance l from the root are in level l.
  - The rest of the proof is an exercise

SHORTEST PATHS FROM ROOT IN BFT

- Graph G:



- BFS(G,1):



OTHER APPLICATIONS OF BFS
-- SHORTEST PATHS WHEN ALL EDGES HAVE SAME WEIGHT (2) --

- Assume all the edges have the same weight (say 1) in a graph G
- **Theorem**: The BFS paths from the root to all the nodes are shortest paths
- **Proof**:
  - Label the levels of the BFT 0, 1, 2, …, top to bottom.
  - Prove by induction the level l that all nodes in level l are of distance l from the root, and that all nodes of distance l from the root are in level l.
  - The rest of the proof is an exercise
- Time to do since-source shortest paths (when edges have same weight): O(|E|), which is faster than the greedy $O(n^2)$ time
- Time to do all-pairs shortest paths (when edges have same weight), by calling BFS n times, one time from each node: O$(n|E|) < O(n^3)$ DP time

LESSONS LEARNED SO FAR

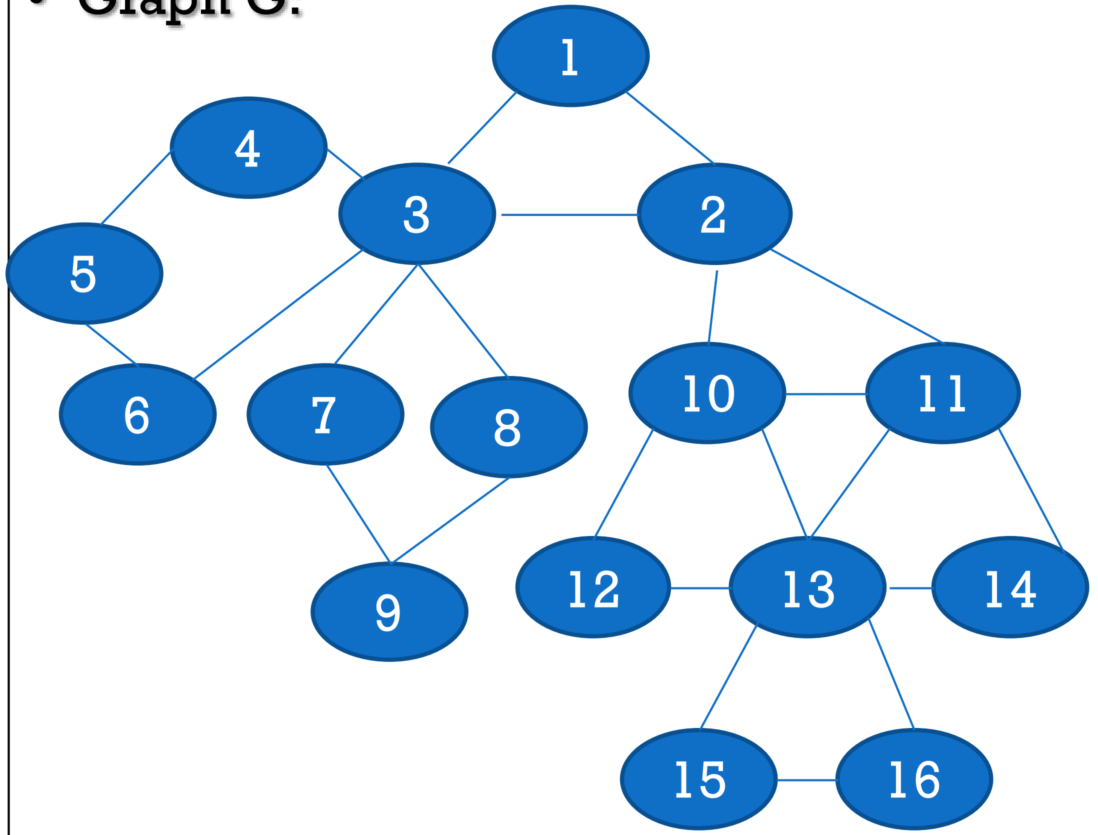
- Tree traversal techniques are simple, recursive, and linear in time
- Sorting a BST is done by applying inorder traversal on it
- Depth-First Search (DFS) and Breadth-First Search (BFS) are generic graph traversal techniques that take linear time (i.e., $O(|E|+|V|)$), and are easily implemented using stacks and queues, respectively
- DFS and BFS have many applications, especially in connectivity, and yield faster algorithms for the MST problem when the edges have the same weight
- BFS yields faster algorithms for the shortest paths problems when the edges have the same weight

BICONNECTIVITY
-- A MAJOR APPLICATION OF DFS --

- **Definition**: A node in a connected graph is called an ***articulation point*** (A.P.) if the deletion of that node (and all the edges incident to it) disconnects the graph.
- **Definition**: A connected graph is called ***biconnected*** if it has no articulation points. That is, the deletion of any single node leaves the graph connected.
- In the case of networks, an articulation point is referred to as a *single point of failure*.

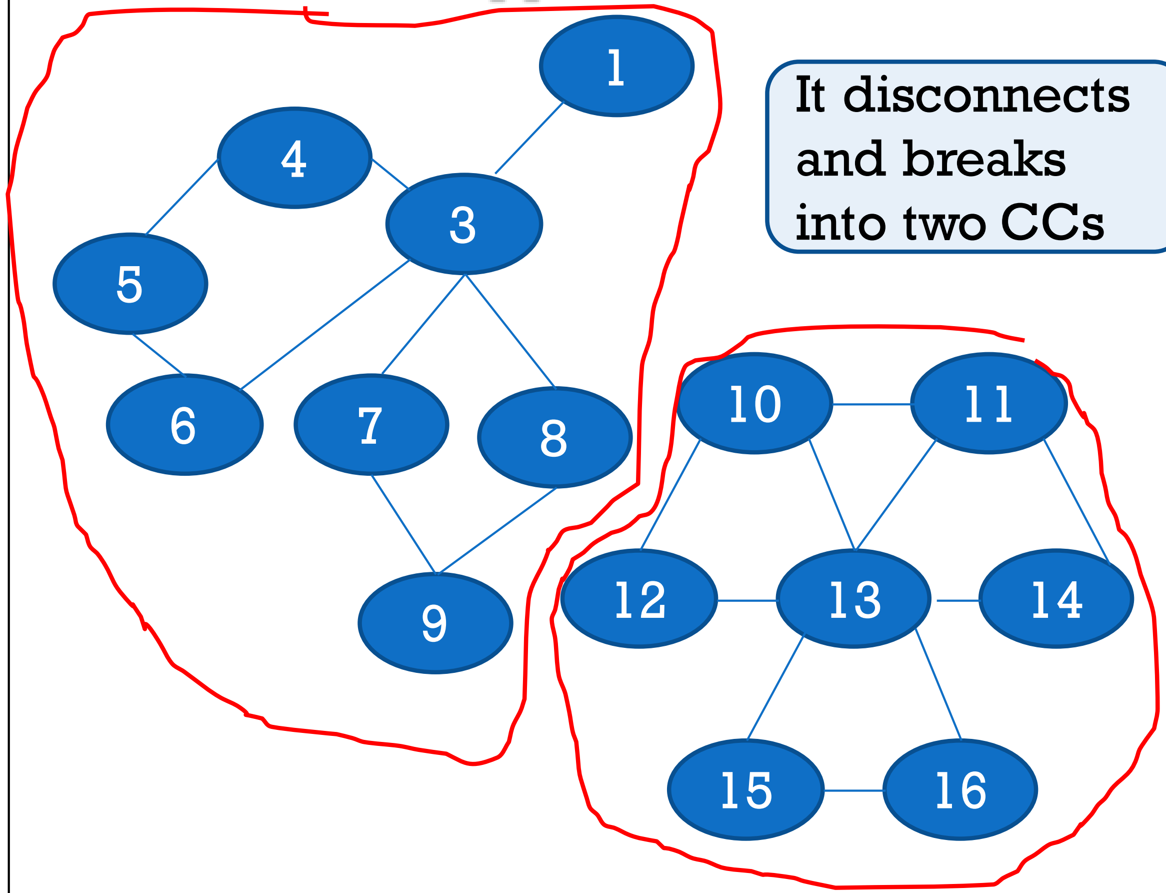
ILLUSTRATION OF ARTICULATION POINTS (1)

- Graph G:



- G is not biconnected
- Nodes 2, 3, and 13 are articulation points

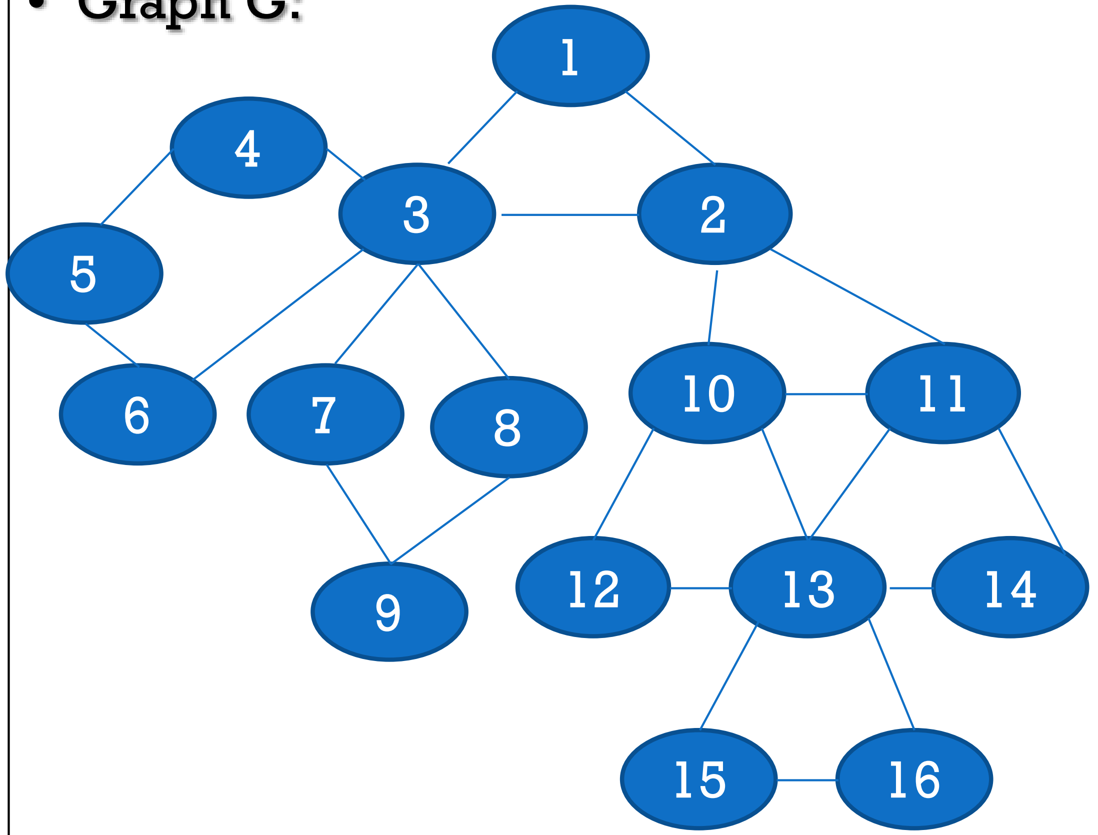
- Observe what happens if we delete node 2:



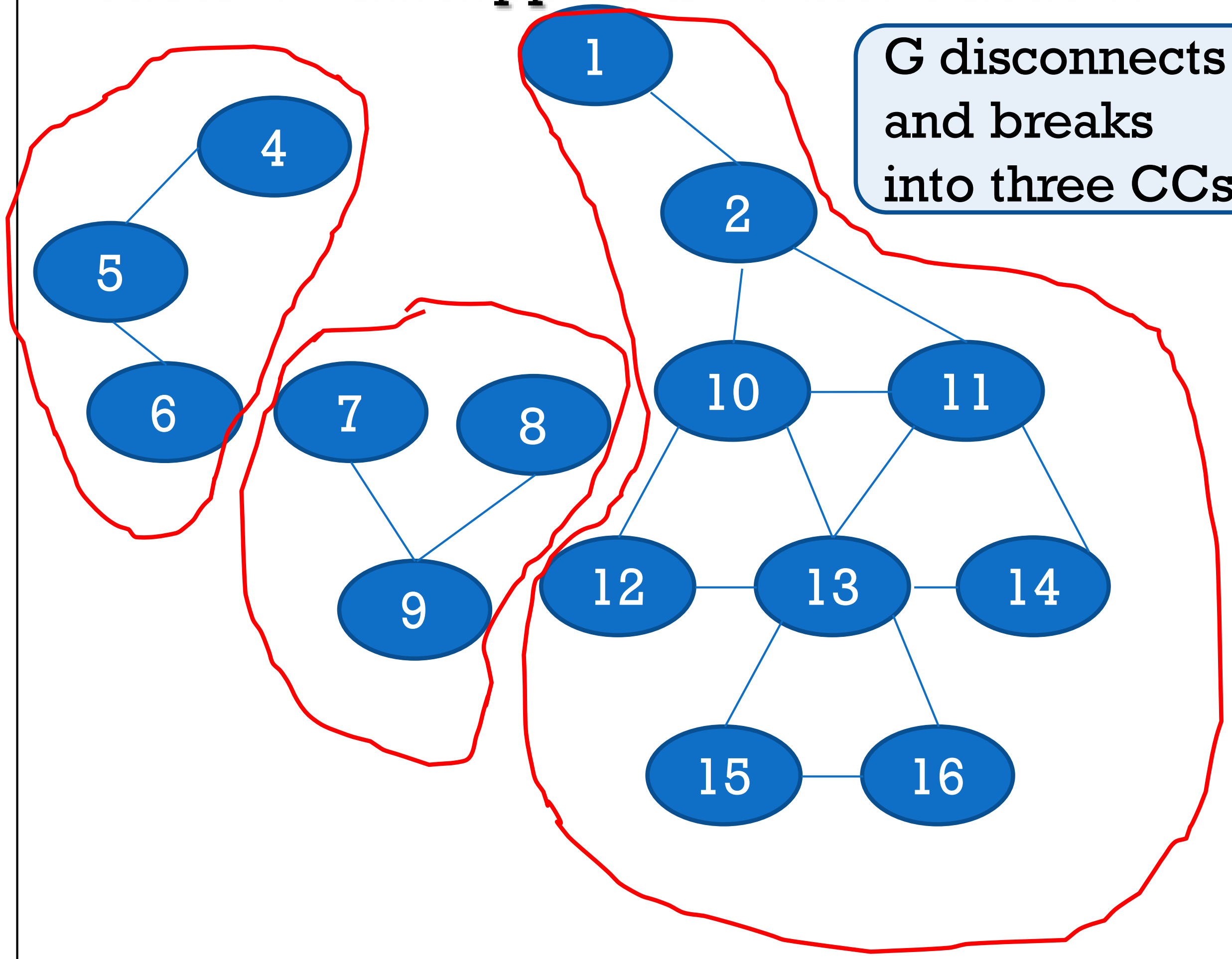
It disconnects and breaks into two CCs

ILLUSTRATION OF ARTICULATION POINTS (3)

- Graph G:



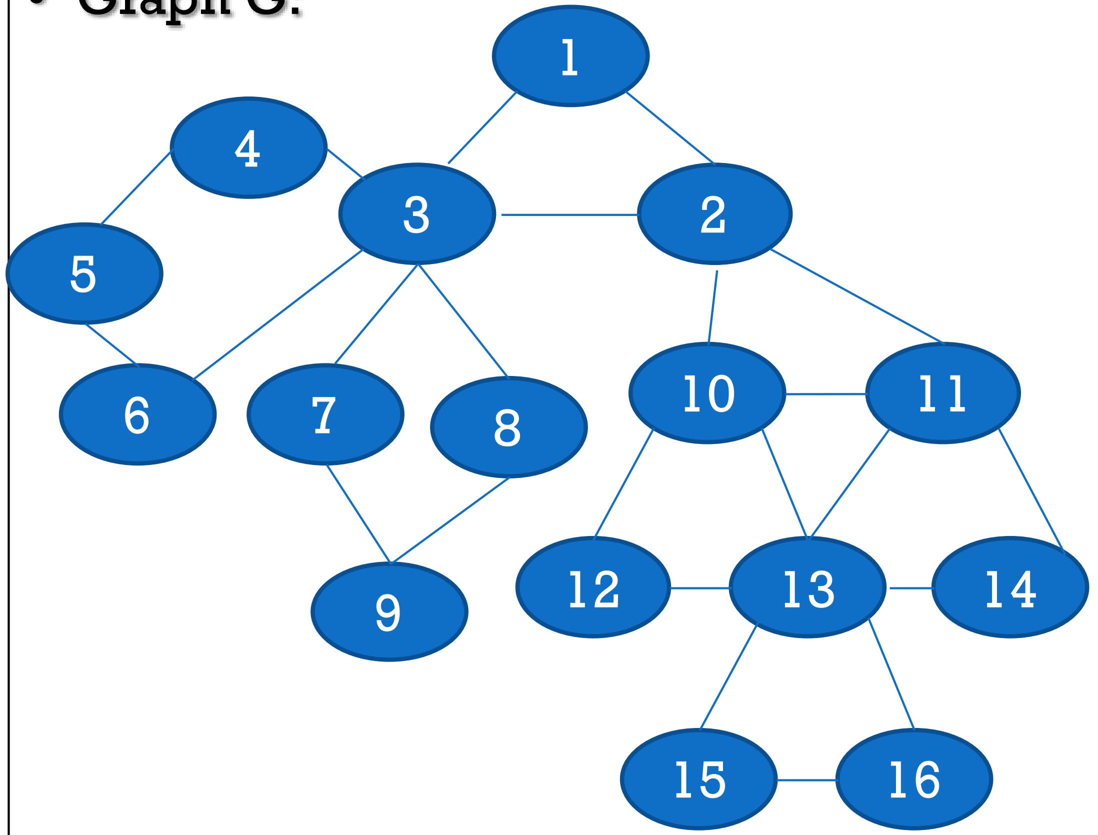
- Observe what happens if we delete node 3:



G disconnects and breaks into three CCs

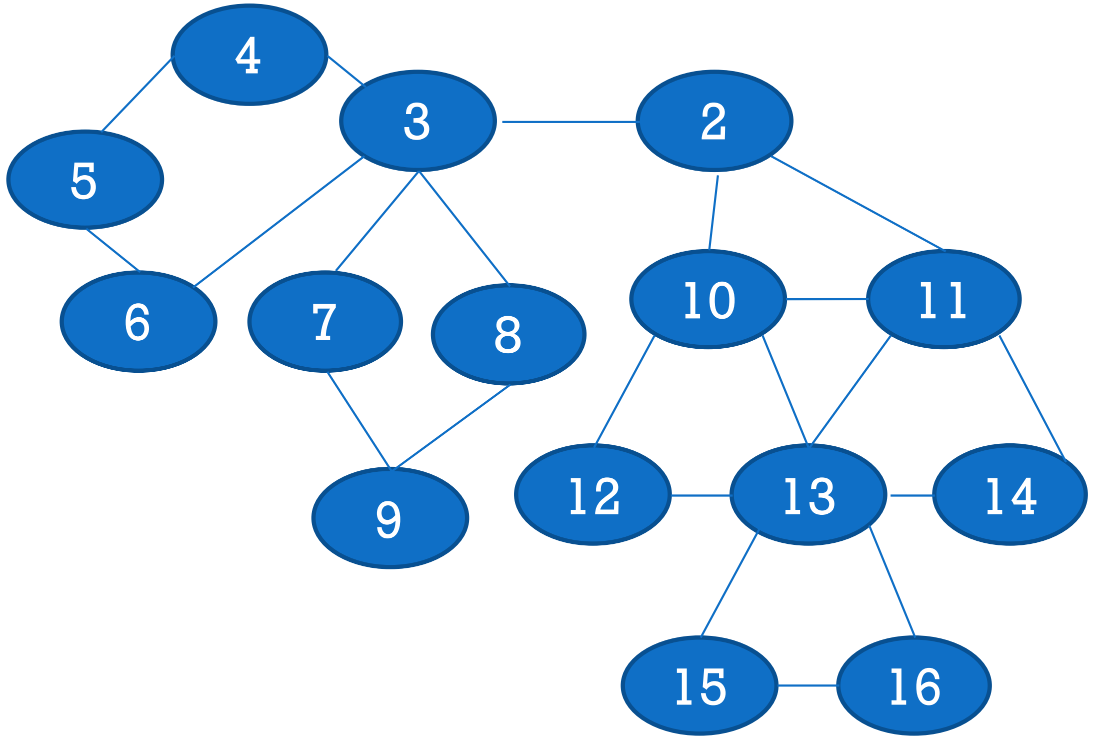
ILLUSTRATION OF NON-ARTICULATION POINTS

- Graph G:



- G is not biconnected
- Nodes 2, 3, and 13 are articulation points

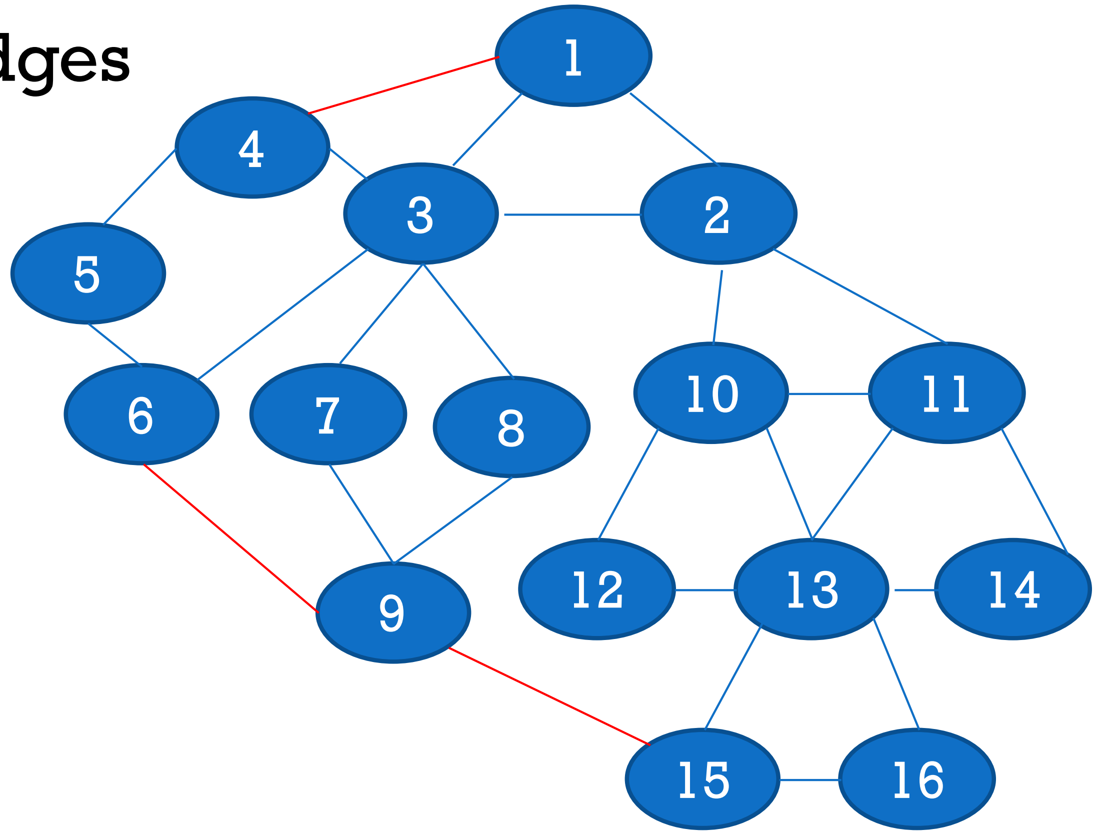
- After deleting node 1, G remains connected
- Hence, node 1 is not an A.P.



BICONNECTIVITY

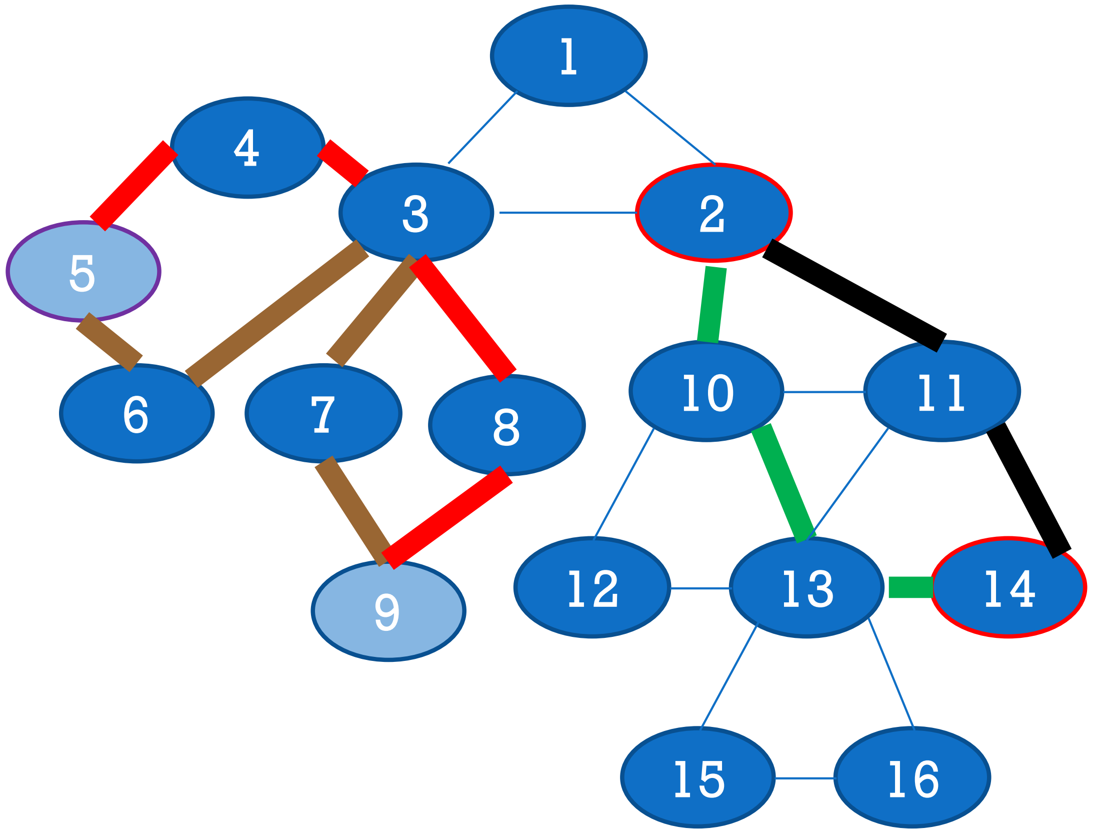
-- AN EXAMPLE OF A BICONNECTED GRAPH --

- The following graph is biconnected: it has no A.P.'s
- Note that by adding some edges (the red ones) to the previous graph, it became biconnected



A BICONNECTIVITY THEOREM

- **Definition**: In a graph G, two paths between a pair of nodes x and y are said to be ***disjoint*** if they don't have any nodes in common except the end-points x and y
- **Theorem**: A graph G is biconnected if and only if between every pair of nodes x and y in G there is at least two disjoint paths
- **Proof**: Will not be covered in this course.



- The black and green paths [2 → 11 → 14] and [1 → 10 → 13 → 14] between 2 and 14 are disjoint
- The red and brown paths [5 → 4 → 3 → 8 → 8] and [5 → 6 → 3 → 7 → 9] between 5 and 9 are not disjoint (they share node 3)

THE BICONNECTIVITY PROBLEM

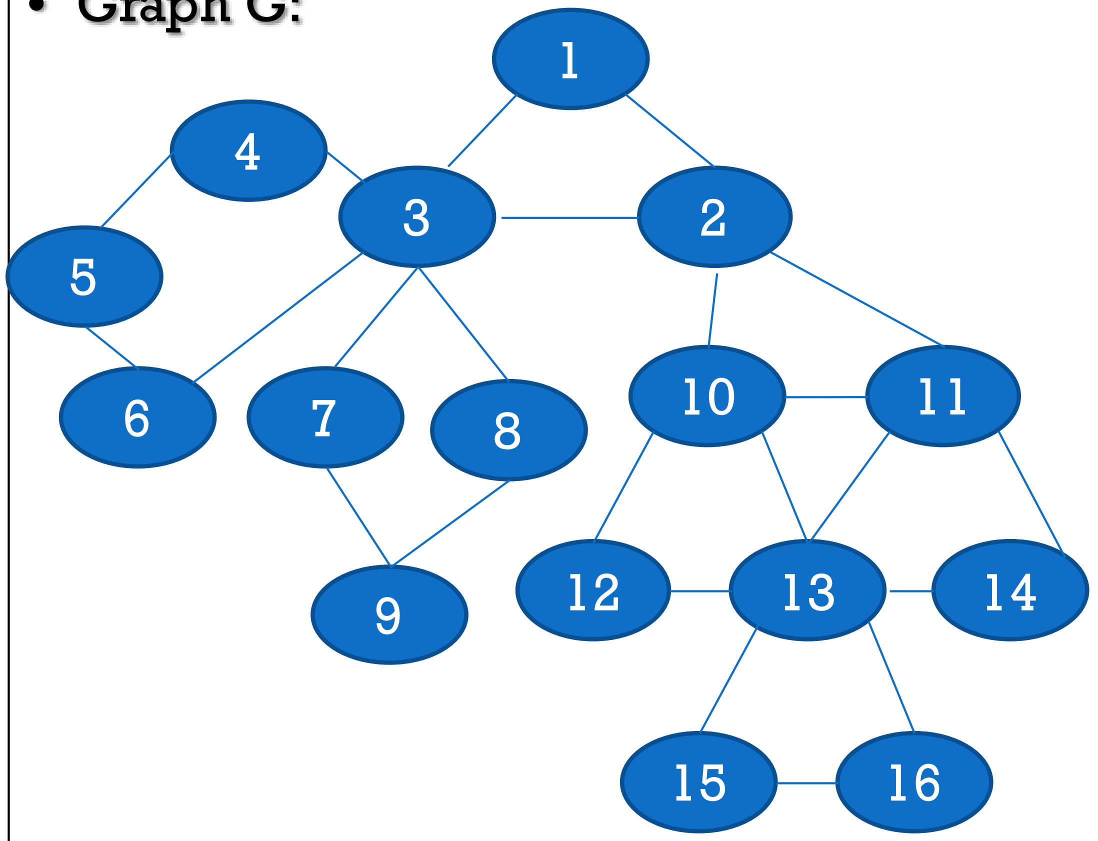
- **Input**: a connected graph G
- **Output**: Whether or not the graph G is biconnected and, if not biconnected, find all the articulation points
- **Task**: Write an algorithm for solving this problem, using a graph traversal technique

BICONNECTIVITY
-- DFS: NEW LAYOUT OF THE GRAPH --

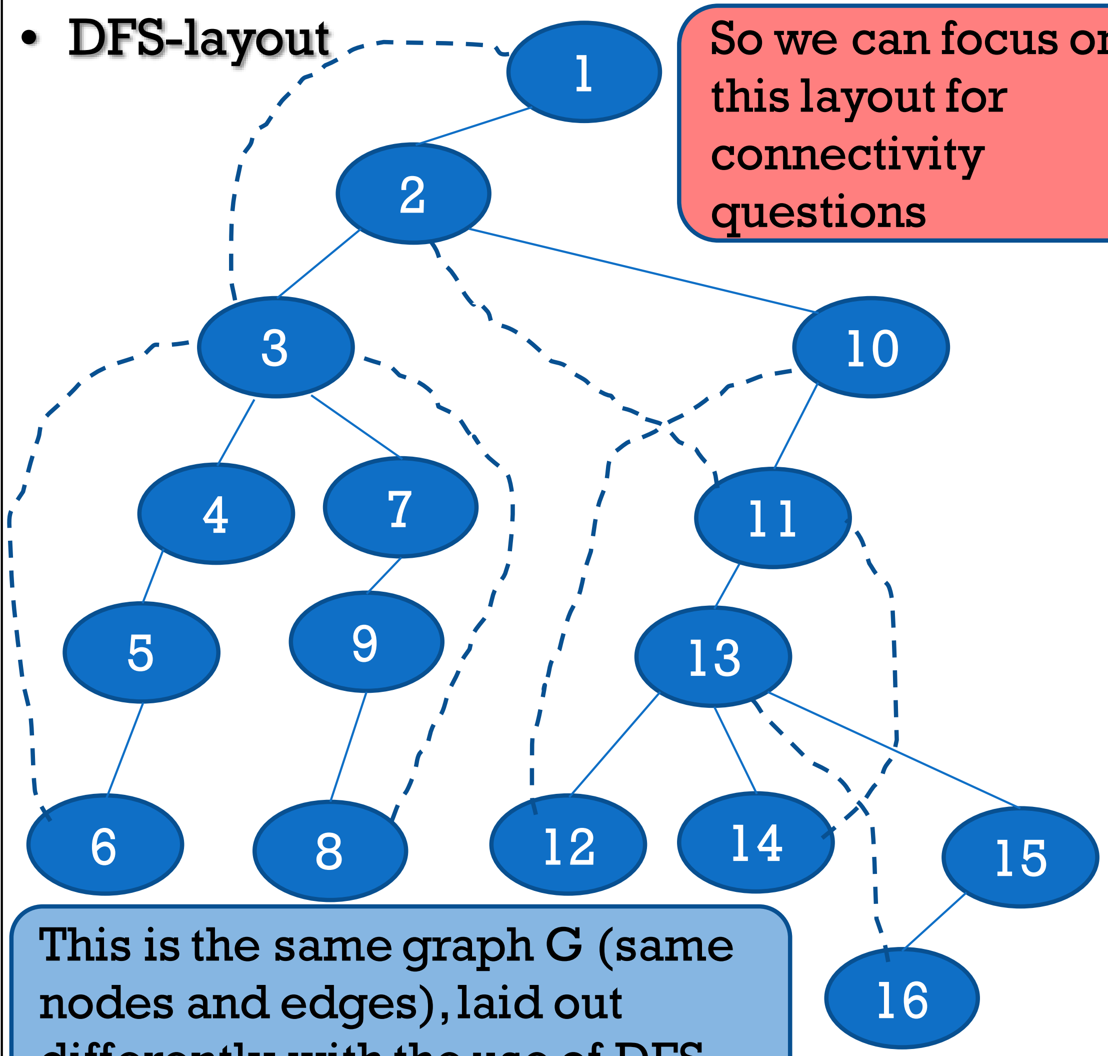
- DFS on a connected graph G yields a DFS tree whose edges are from the graph.
- Draw those edges as **solid** edges.
- Add the remaining edges of the graph as **dashed** edges in the tree.

DFS-LAYOUT OF A GRAPH

- Graph G:



- DFS-layout

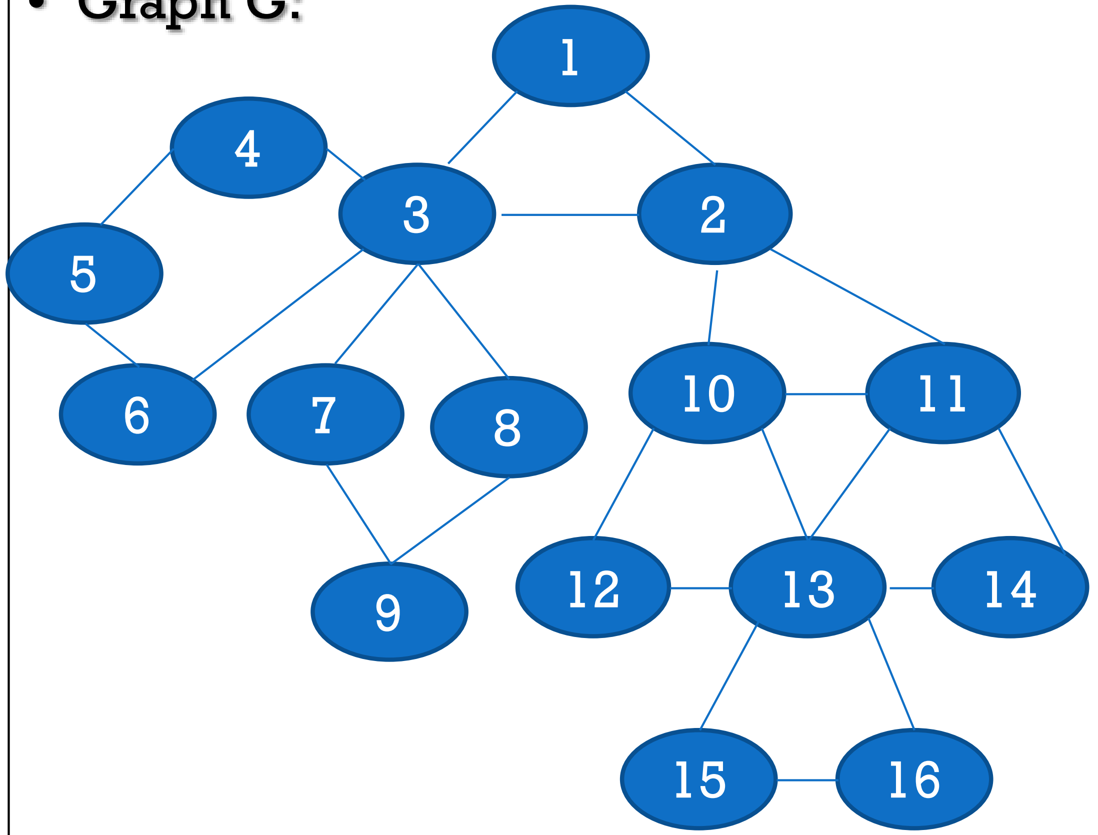


So we can focus on this layout for connectivity questions

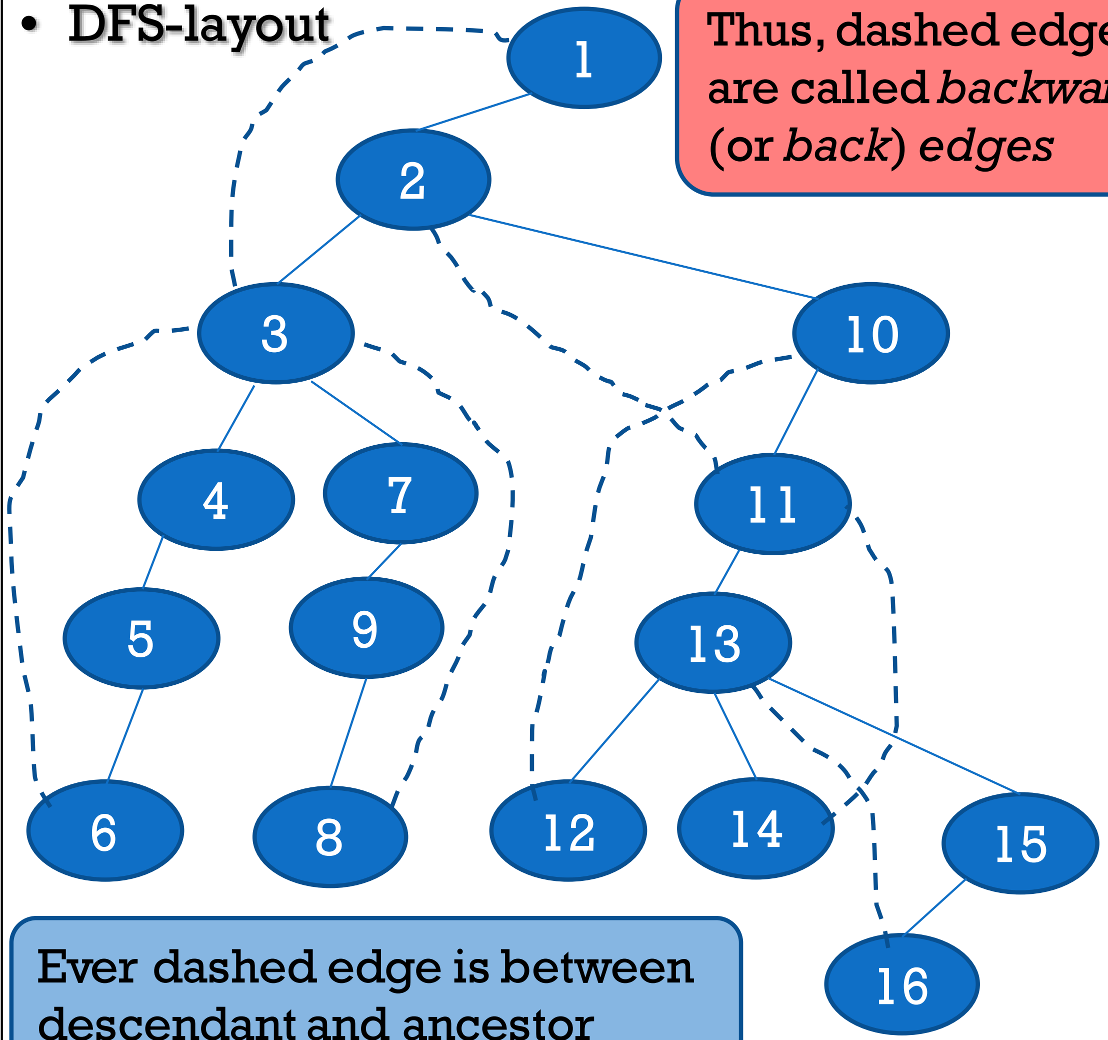
This is the same graph G (same nodes and edges), laid out differently, with the use of DFS

OBSERVATIONS ABOUT THE DASHED EDGES

- Graph G:



- DFS-layout

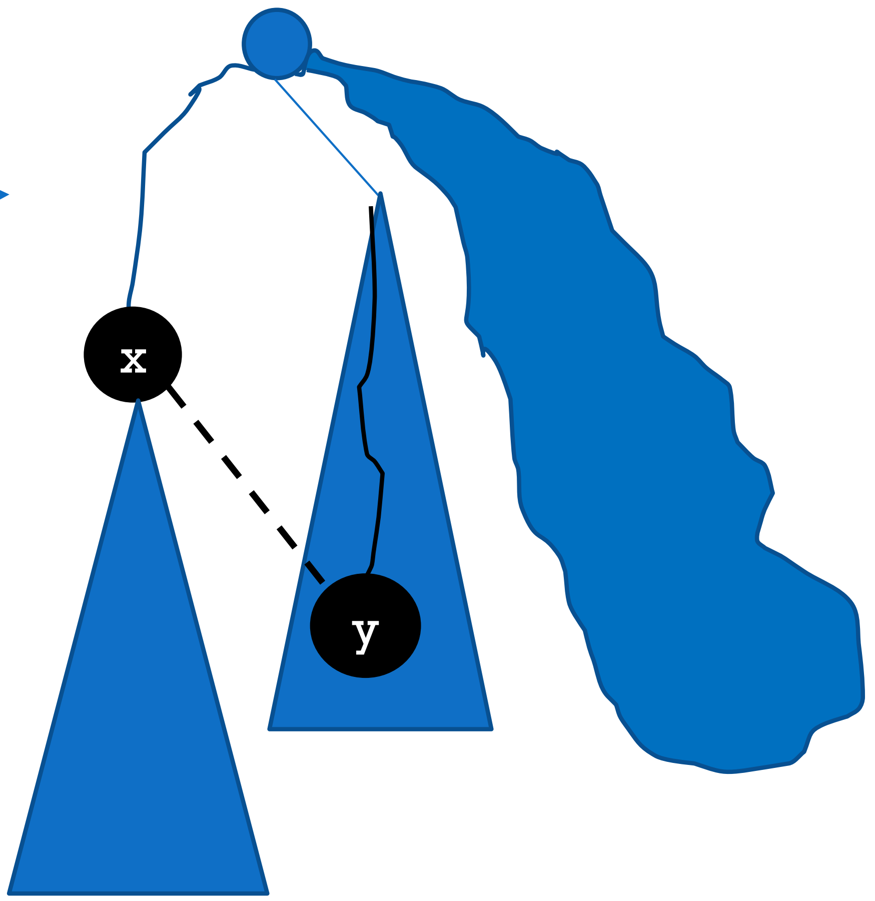


Thus, dashed edge are called *backward* (or *back*) *edges*

Ever dashed edge is between descendant and ancestor

BACKWARD EDGES IN DFS -- A THEOREM --

- **Theorem**: Each dashed edge in a DFS layout goes from a descendant to an ancestor.
- **Proof**: The proof is by contradiction.
  - Take a graph G and do a DFS on it
  - Let (x,y) be a dashed edge between nodes that are not ancestor-descendant, that is, x and y are in separate subtrees of the DFS tree. Assume x was visited before y.
  - Let t be the time DFS backtracks from x: no unvisited neighbors of x remain, and DFS will not return to x again
  - Since x is visited before y and y is not a descendant of x, **y has not been visited at time t**.
  - But since y is a neighbor of x and y is not visited at time t, y would have to be visited from x before the algorithm backtracks from x.
  - That would make y a descendant of x. Contradiction.
  - Therefore, no such cross edge (x,y) can exist in a DFS tree. Q.E.D.

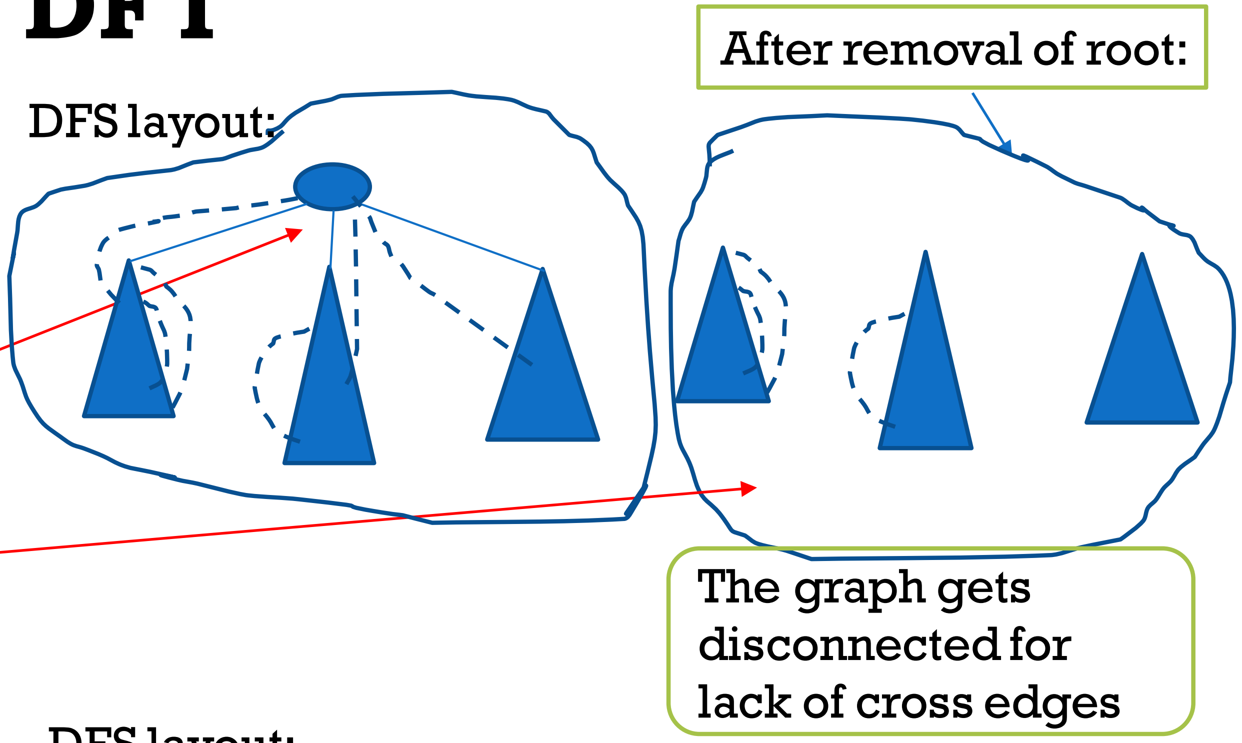


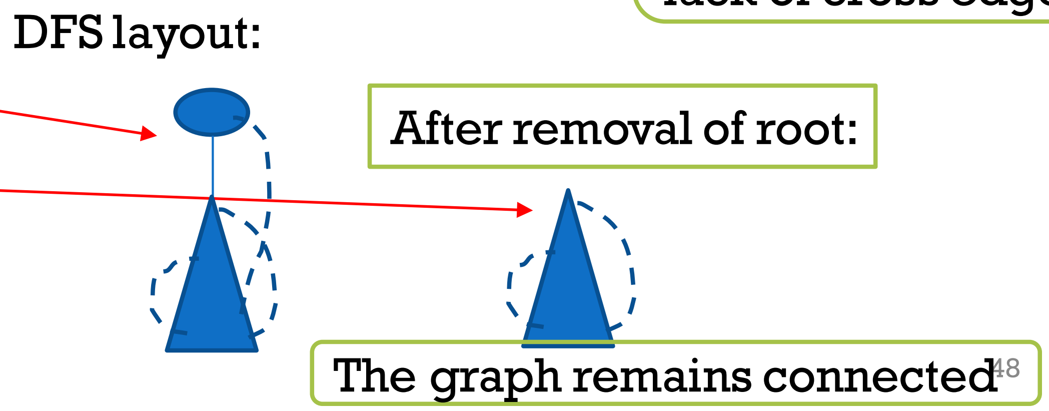
LESSONS LEARNED SO FAR

- Tree traversal techniques are simple, recursive, and linear in time
- Sorting a BST is done by applying inorder traversal on it
- Depth-First Search (DFS) and Breadth-First Search (BFS) are generic graph traversal techniques that take linear time (i.e., O(|E|+|V|)), and are easily implemented using stacks and queues, respectively
- DFS and BFS have many applications, especially in connectivity, and yield faster algorithms for the MST problem when the edges have the same weight
- BFS yields faster algorithms for the shortest paths problems when the edges have the same weight
- DFS results in a new layout of graphs (tree edges and back edges) that is more illuminating about the underlying structure of the graph than any random layout

OBSERVATIONS ABOUT THE ROOT OF THE DFT

- **Theorem**: If the root has more than one child, then the root is an articulation point. But if the root has a single child, it is not an articulation point.
- **Proof**:
  - Case 1: the root has multiple children
    - the removal of the root makes the subtrees of that root disconnected from one another since there are no cross dashed edges between them. (see top half of figure)
  - Case 2: the root has one child
    - its removal leaves the remaining n-1 nodes connected by at least the (only) subtree of the root.   Q.E.D.



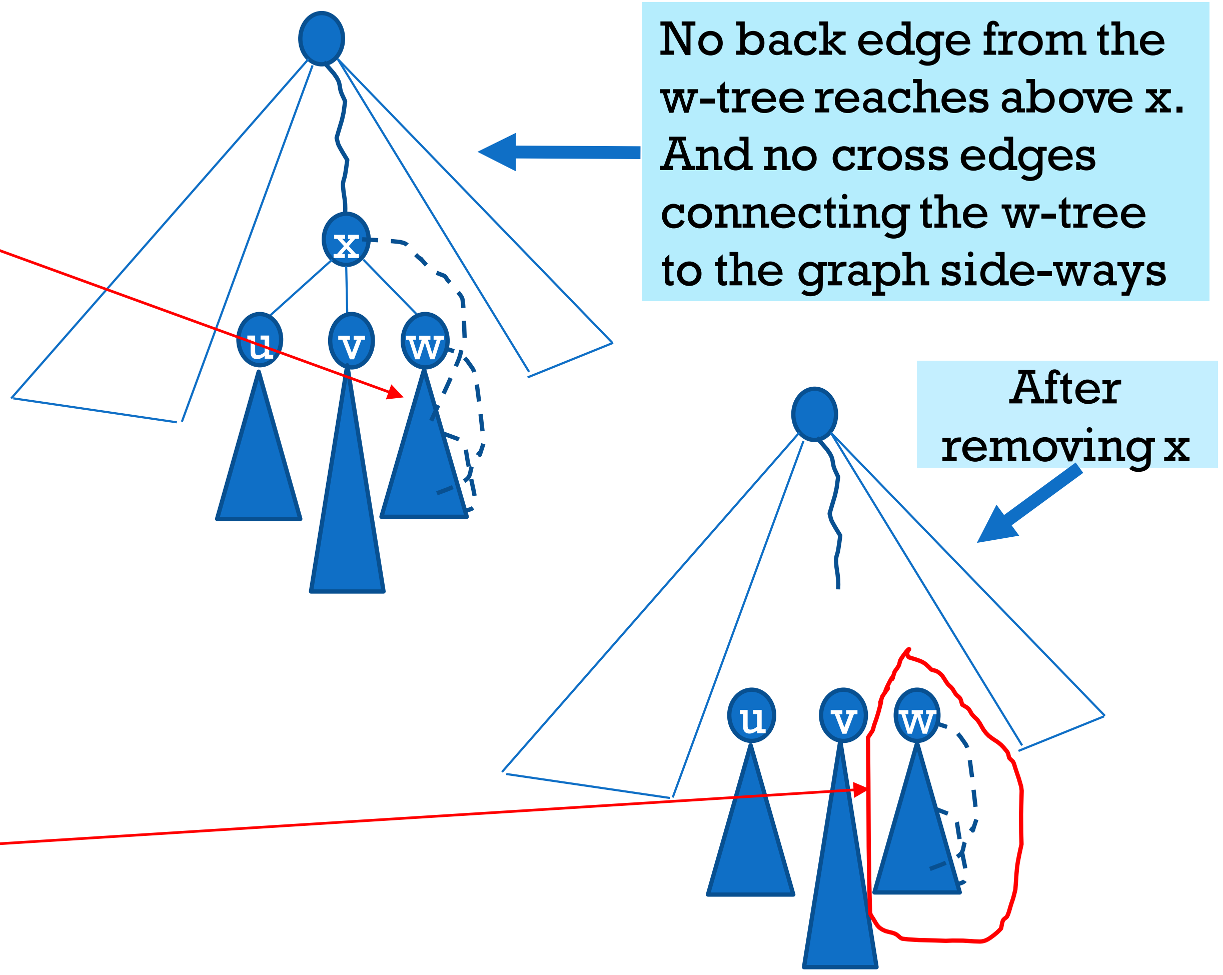


CAN THIS LEAD TO AN ALGORITHM FOR FINDING ALL AP'S?

- The previous theorem suggests a first algorithm for identifying articulation points:
  - For each node x in G, do a DFS from x, and check if x more than one child. If so, x is an a.p.
  - This algorithm, however, takes $O(n\ |E|)$ **time** (b/c it calls DFS n times)
- Can we do better?
- A better algorithm will be designed that takes only O(|E|) time, performing only one DFS instead of n DFSs

TESTABLE CRITERION FOR A NON-ROOT TO BE A.P.

- Notation: Let ***w-tree*** denote the subtree rooted at w in a DFT
- Necessary and sufficient condition for a non-root to be an A.P.:
    - **A non-root node x is an articulation point** if and only if **x has a subtree (w-tree) such that every back edge that originates from that substree ends at x or at w or at a descendant of w**
- Why? Observe in the figure how when x is removed, the w-tree is diconnected from the rest of the graph


No back edge from the w-tree reaches above x. And no cross edges connecting the w-tree to the graph side-ways
x
u
v
w
After removing x
u
v
w

REFINING THE CRITERION FOR A NON-ROOT TO BE A.P.

- Let $L[w] \overset{\text{def}}{=}$ the highest node reachable (from the w-tree by a back edge) or (from w by a null path)
- The AP criterion on the previous slide "**x has a subtree (w-tree) such that every back edge that originates from that substree ends at x or at w or at a descendant of w**"

  becomes:

  "x is an articulation point iff x has a child w where L[w] is x or w or a descendant of w"
- We will turn this "visual/geometric" criterion into a numerical criterion

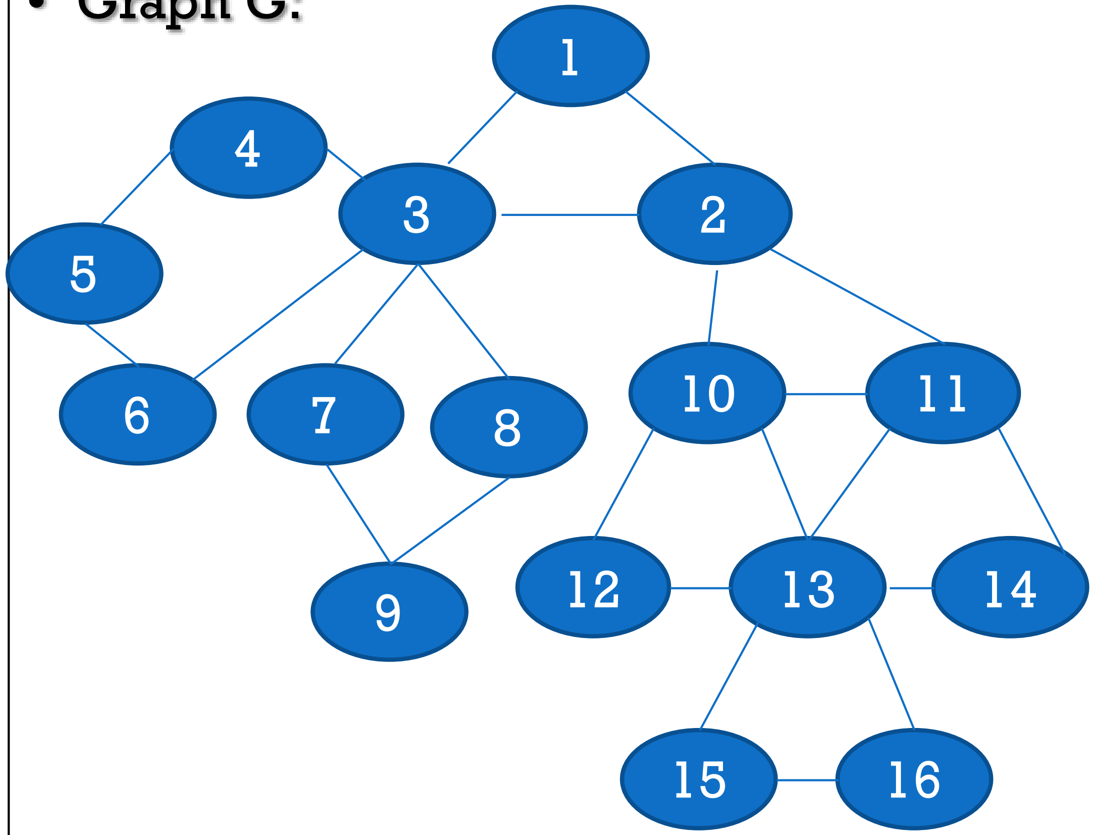
QUANTIFYING THE AP CRITERION

-- RELABELING THE NODES TO CAPTURE GEOMETRY (1) --

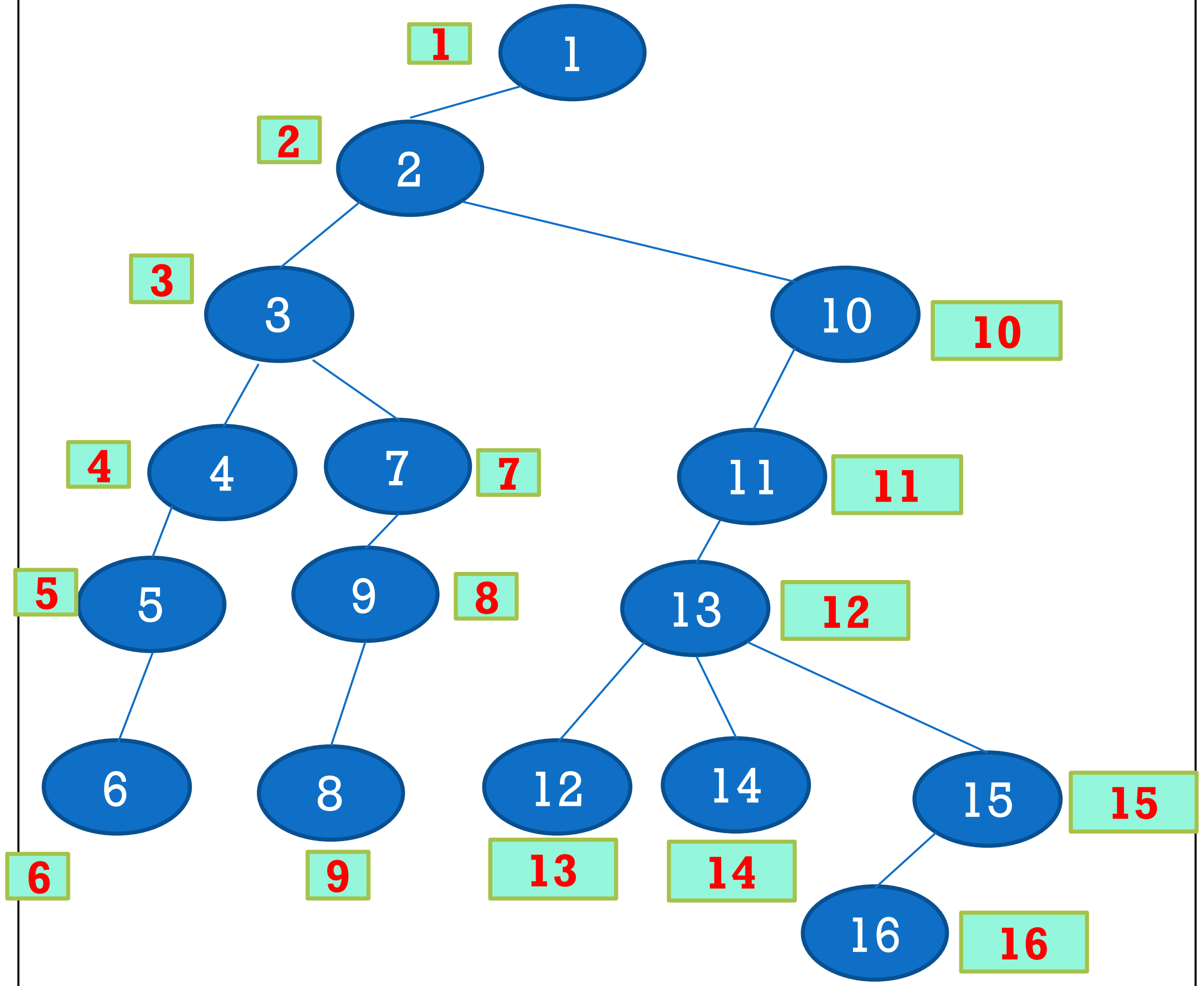
- The nodes of the graph will be **relabeled** so that the new labels carry meaningful information.
- Indeed, each node i will have two new labels: $DFN[i]$ and $L[i]$.
- Let $\boldsymbol{DFN[i]} \stackrel{\text{def}}{=}$ **the time at which i is visited in DFS**. Thus, the 1st node visited (the root) has its DFN $= 1$. The 2nd node visited has a DFN $= 2$, etc.

ILLUSTRATION OF DFN RELABELING

- Graph G:



DFS with DFN labels inside green boxes:



QUANTIFYING THE AP CRITERION
-- RELABELING THE NODES TO CAPTURE GEOMETRY (2) --

- Recall $L[w] \stackrel{\text{def}}{=}$ the highest node reachable (from the w-tree by a back edge) or (from w by a null path)

- Express $L[w]$ **in terms of DFN:**

$L[w] \stackrel{\text{def}}{=}$ **the DFN of [the highest node reachable (from the w-tree by a back edge) or (from w by a null path)]**

- The AP criterion "x is an articulation point iff x has a child w where L[w] is x or w or a descendant of w" becomes:

"x is an articulation point iff x has a child w where L[w] is DFN[x] or DFN[w] or DFN[a descendant of w]"

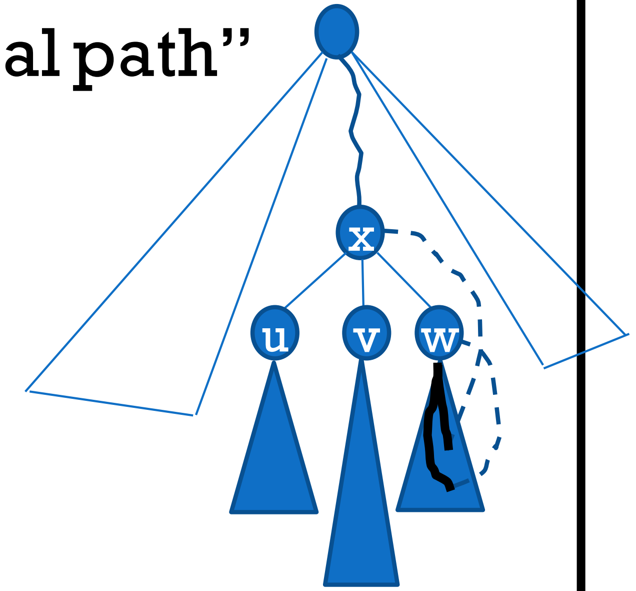
QUANTIFYING THE AP CRITERION
-- IN TERMS OF L AND DFN --

- Observe that
  - the ancestors of w have DFN's $\leq \mathrm{DFN}[w]$
  - The descendants of w have DFN's $\geq \mathrm{DFN}[w]$
- The AP criterion becomes "x is an articulation point iff x has a child w where L[w] is DFN[x] or DFN[w] or DFN[a descendant of w]" becomes

  x is an articulation point iff x has a child w where $L[w] \geq DFN[x]$

- Therefore, if we can compute the DFN's and L's of all the nodes, we can apply that criterion at every node x to identify the articulation points

HOW TO COMPUTE *L*[*W*]
-- USING THE NOTION OF SPECIAL PATHS (1) --

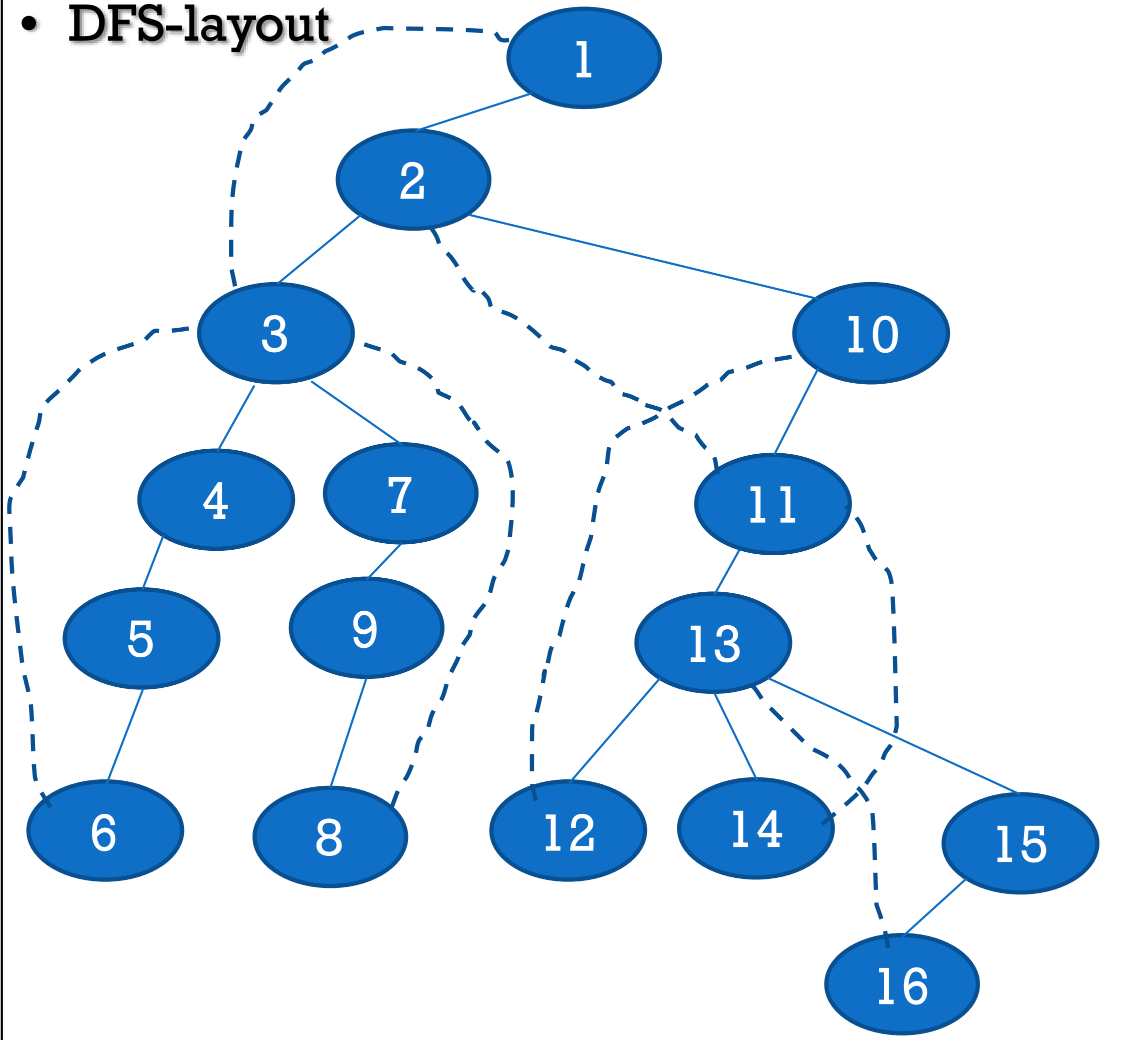


- For convenience, we'll define the notion of "special path"
- A ***special path*** from a node *w* is
  - the null path from *w* to itself, or
  - A path that goes from *w* **downward zero or more <u>tree</u> edges**, and ends with one upward **<u>back</u>** edge
- Therefore, every node reachable from the *w*-tree with a back edge is reachable from w with a special path
- This observation will help us state L[*w*] in terms of special paths

ILLUSTRATION OF SPECIAL PATHS

- Special paths from node 11:
  - **Null path 11**
  - **Path 11 → 2 (one upward back edge)**
  - **11 → 13 → 12 →10: it went down tree edges from 11 to 13 to 12, and finally up a back edge to 10**
  - **11 → 13 → 14 →11**
  - **11 →13 →15 →16 →13**
- The highest node reachable by any of those special paths from 11 is node 2
- DFN[2]=**2** (see a couple of slides ago)
- Therefore, L[11]=**2**

- DFS-layout



HOW TO COMPUTE $L[W]$ -- USING THE NOTION OF SPECIAL PATHS (2) --

1. Recall $L[w] \overset{\text{def}}{=}$ the DFN of the highest node reachable from the w -tree by a back edge or from w by a null path
2. It can be easily seen that

   $L[w]$ = the DFN of the highest node reachable from w by a special path
3. Observe that the highest node reachable from w by a special path is either w itself or a proper ancestor of w. Either way, it is an ancestor of w.
4. Observe also that in the set of ancestors of w, the higher (geometrically) an ancestor is, the smaller its DFN
5. By the last 3 points, we conclude that

   $L[w]$ = min{DFN[x] | x is reachable from w by a special path}

HOW TO COMPUTE $L[W]$ -- USING THE NOTION OF SPECIAL PATHS (3) --

- We can divide the special paths from w into three groups:
  1. **Group 1: The null path (reaching w)**
  2. **Group 2: the paths made up of a single upward back edge from w**
  3. **Group 3: the special paths that go down at least one tree edge before proceeding**
- Group 3 can be viewed as all non-null special paths from the children of w
- Even if we include the null paths (from the children of w) in group 3, that won't change the value of $L[w]$
- Thus, the groups of special paths from w can be restated as:
  1. **Group 1: The null path (reaching w)**
  2. **Group 2: the paths made up of a single upward back edge from w**
  3. **Group 3: the special paths from the children of w**

HOW TO COMPUTE $L[W]$
-- USING THE NOTION OF SPECIAL PATHS (4) --

- Using $L[w] = \min\{\mathrm{DFN}[x] \mid x \text{ is reachable from } w \text{ by a special path}\}$

  and the three groups of special paths from w, we conclude:

$$L[w] = \min\{\ \min\{\mathrm{DFN}[x] \mid x \text{ in Group 1}\},\ \min\{\mathrm{DFN}[x] \mid x \text{ in Group 2}\},\ \min\{\mathrm{DFN}[x] \mid x \text{ in Group 3}\}\ \}$$

- **$\min\{\mathrm{DFN}[x] \mid x \text{ in Group 1}\} = \mathrm{DFN}[w]$**
- **$\min\{\mathrm{DFN}[x] \mid x \text{ in Group 2}\} = \min\{\mathrm{DFN}[x] \mid (w,x) \text{ is an upward back edge}\}$**
- For Group 3, divide into subgroups, one subgroup per child v of w
  - Therefore, the min from Group 3 is the min of mins of the subgroups
  - But the min of a subgroup corresponding to a child v of w is $L[v]$
  - We conclude: **min of Group 3 = $\min\{L[v] \mid v \text{ is a child of } w\}$**
- Hence

$$L[w] = \min\{\mathit{DFN}[w], \min\{\mathrm{DFN}[x] \mid (w,x) \text{ is an upward back edge}\}, \min\{L[v] \mid v \text{ is a child of } w\}\}$$

CALCULATIONS OF *L*[*W*]
-- BOTTOM UP --

- The final formula we got for L[w] needs the L of its children
- So the L's of the nodes have to be computed bottom up (i.e., from the leaves upward)
- Observe that if w is a leaf, its Group 3 is empty, and thus its
  $L[w] = \min\{DFN[w], \min\{DFN[x] \mid (w,x) \text{ is an upward back edge}\}\}$
- Therefore, for leaves, the computations of the L is relatively easy
- The computations of L is illustrated next

ILLUSTRATIONS OF COMPUTING THE L'S

Using the DFN labels in green squares, compute L labels inside black circles :

- L[6]=min{DFN[6], DFN[3]}=min{6,3}=3
- L[5]=min{ DFN[5], L[6]}=min{5,3}=3
- L[4]=min{DFN[4], L[6]}=min{4,3}=3
- L[8]=min{DFN[8],DFN[3]}=min{9,3}=3
- L[9]=min{DFN[9], L[8]}=min{8,3}=3
- L[7]=min{DFN[7], L[9]}=min{7,3}=3
- L[3]=min{DFN[3], DFN[1]

  b/c (3,1) is a back edge,

  min{L[4], L[7]}}

  = min{3,1,3,3}=1
- Etc.

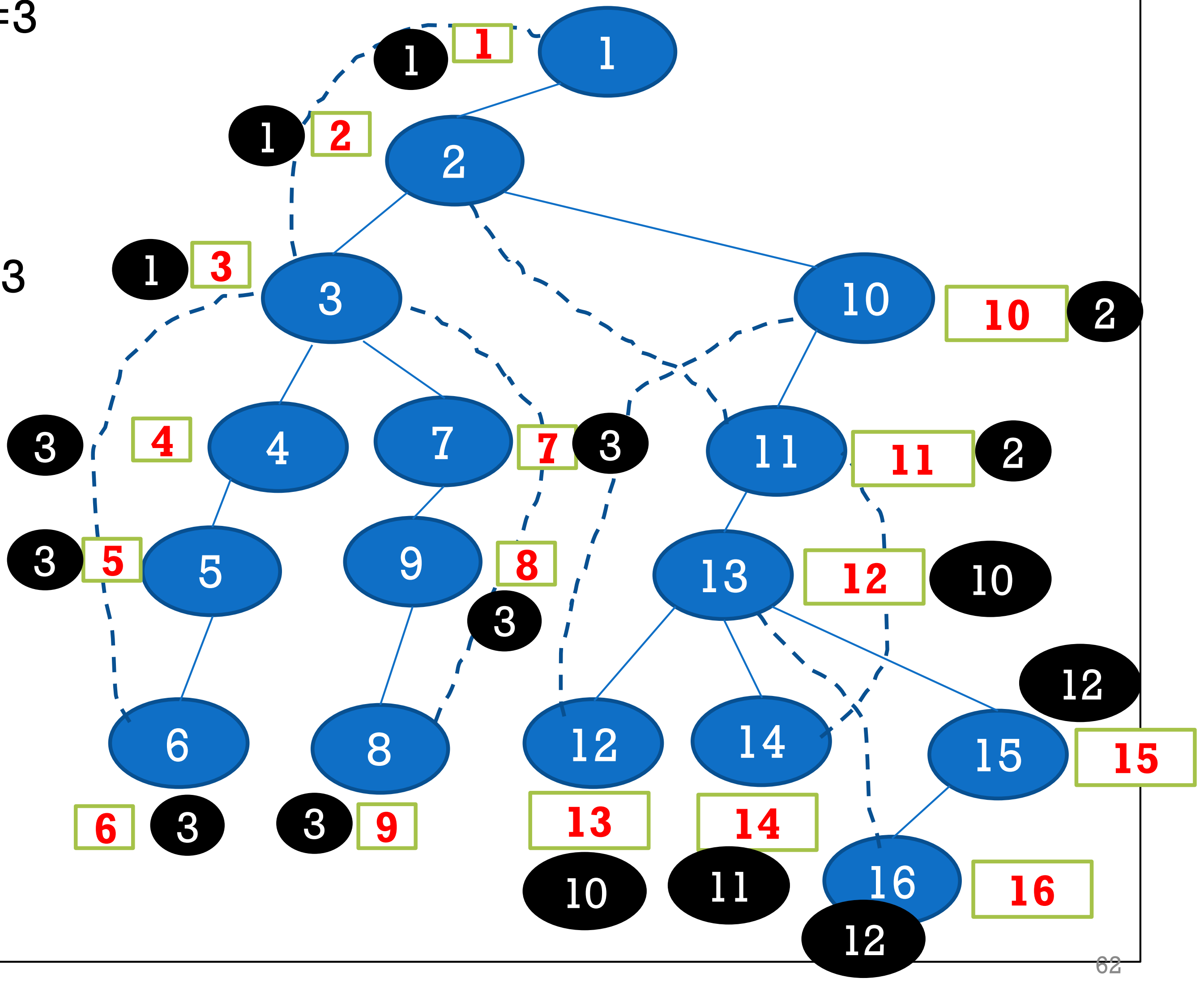


ILLUSTRATION OF IDENTIFYING THE AP'S

AP criterion: x is an articulation point iff x has a child w where $L[w] \geq DFN[x]$

- Do a DFS, & compute the DFN and L of all the nodes
- Apply the AP criterion at every node

  x is an articulation point iff

  x has a child w where $L[w] \geq DFN[x]$

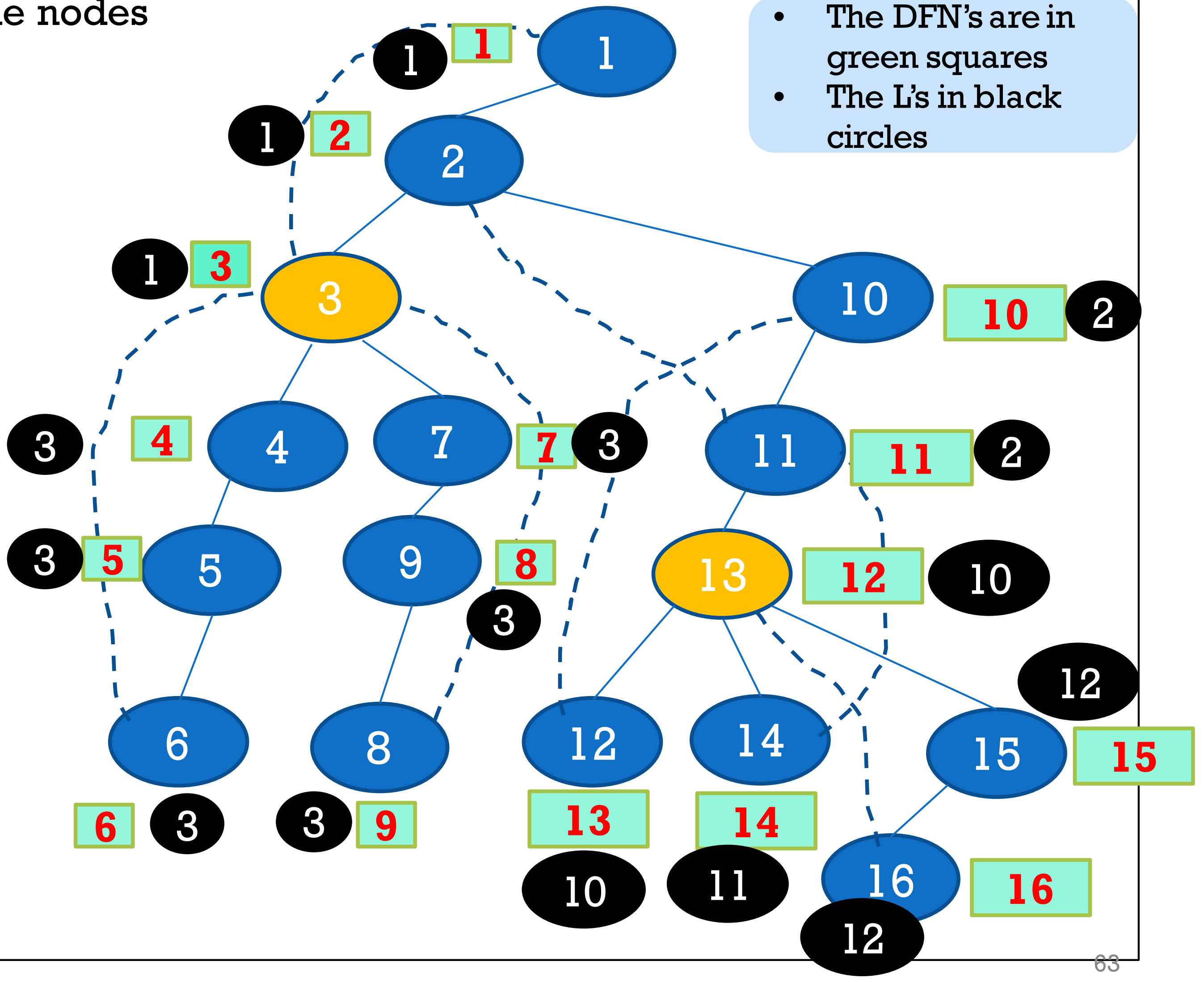
- 3 has a child 4 where L[4] ≥DFN[3],

  i.e., 3 ≥3, thus **3 is AP**
- 13 has a child 15 where L[15] ≥DFN[13],

  i.e., 12 ≥12, thus **13 is AP**
- 2 has a child 10 where L[10] ≥DFN[2],

  i.e., 2 ≥2, thus **2 is AP**
- The AP criterion is not satisfied at any other node
- Therefore, the APs are: 2, 3, 13

- The DFN's are in green squares
- The L's in black circles



DFS-BASED ALGORITHM FOR ARTICULATION POINTS

-- THE ADDED PIECES OF CODE ARE HIGHLIGHTED --

```
Procedure DFS(input: graph G) // for articulation points
begin
    Stack S;
    int x, y, v;
    int DFN[1:n], L[1:n], Parent[1:n], num := 1;
    v := an unvisited node; // a starting node
    visit(v);        push(v,S);
    DFN[v] := num; num++; L[v] := DFN[v];
    while (S is not empty) do
        x := top(S); // current node
        if (x has an unvisited neighbor y) then
            visit(y); push(y,S);
            DFN[y] := num; num++; Parent[y] := x;
            L[y] := DFN[y];
        else
            pop(S); // backtrack to previous node
        endif
    endwhile
end
```

```
for (every neighbor y of x) do
    if (y != parent[x] and DFN[y] < DFN[x])
    then
        // y is an ancestor of x, and
        // (x,y) is a back edge
        L[x] := min(L[x],DFN[y]);
    else
        if (x = Parent[y]) then
            L[x] : min(L[x],L[y]);
            if (L[y] >= DFN[x] && x not
            root) then
                x is an articulation point;
            endif
        endif
    endif
endfor
```

```
if (v has more than one child) then
    v is an articulation point;
endif
```

BICONNECTIVITY
-- TIME COMPLEXITY --

- The new statements add constant-time operations except for the new for loop at the time of backtracking
- This new for-loop crosses the edges one more time to update the L values and check for articulation points
  - This increases the time by another O($|E|$)
- The final if-statement, to check for the status of the root, can be done by scanning the Parent array to count the number of children of the root v
  - By counting the number of nodes whose Parent is v)
  - It takes O(n) = O($|E|$) time (b/c G is connected, and so $|E| \geq n - 1$)
- Therefore, the time complexity of the whole algorithm is O($|E|$).

LESSONS LEARNED SO FAR

- Tree traversal techniques are simple, recursive, and linear in time
- Sorting a BST is done by applying inorder traversal on it
- Depth-First Search (DFS) and Breadth-First Search (BFS) are generic graph traversal techniques that take linear time (i.e., O(|E|+|V|)), and are easily implemented using stacks and queues, respectively
- DFS and BFS have many applications, especially in connectivity, and yield faster algorithm for the MST problem when the edges have the same weight
- BFS yields faster algorithms for the shortest paths problems when the edges have the same weight
- DFS results in a new layout of graphs (tree edges and back edges) that is more illuminating about the underlying structure of the graph than any random layout
- Clever use of DFS and its illuminating layout of graphs enable the derivation of very efficient, sophisticated algorithms for advanced connectivity problems (e.g., biconnectivity and articulation points)
- Quantification of symbolic/geometric notions, wherever possible, leads to elegant and/or efficient algorithms

OTHER APPLICATIONS OF GRAPH TRAVERSAL

- Planarity testing
  - A graph is said to be planar if it can be laid out in such a way that no two edges cross at non-nodes
  - DFS has been used to derive an efficient algorithm for planarity testing
- K-connectivity: this is a generalization of biconnectivity: a graph is K-connected if the simultaneous removal of any K-1 nodes leaves the graph connected
  - DFS has been used to derive an efficient algorithm for testing if an input graph is K-connected

OTHER GRAPH TRAVERSAL PROBLEMS -- OPTIONAL --

- Modify DFS and DFS to apply to directed graphs
- Recall strong connectivity of directed graphs (a digraph is strongly connected if every node is reachable by path from a every other node)
  - Write a directed-DFS algorithm to check if a digraph is strongly connected
- An edge in an undirected graph is called a ***bridge*** if the removal of the edge (but not its nodes) disconnects the graph.
  - Develop a DFS-based algorithm to identify all the bridges in an input graph (Hint: use the notion of L and DFN, and derive a bridge criterion similar to the AP criterion)