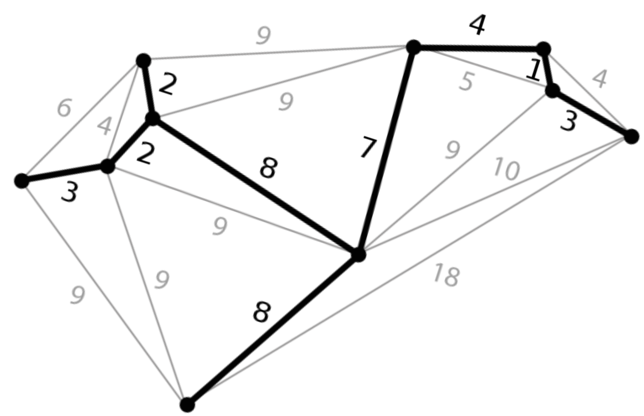
COMP261
Algorithms and Data Structures
2024 Tri 1

Jyoti Sahni
jyoti.sahni@ecs.vuw.ac.nz
Office Hours (COMP261): AM414, Thursday 10:00 – 12:00

Recap: Spanning Trees

Given a connected, undirected, weighted graph, a spanning tree is a subgraph that contains all the nodes but has no cycles (is a tree)

A spanning tree is defined only for a connected graph, because a tree is always connected, and in a disconnected graph of n vertices we cannot find a connected subgraph with n vertices



Recap: Spanning trees in Weighted graphs

The spanning-tree problem

- Add nodes to partial tree
- Add acyclic edges

Minimum-cost-spanning-tree problem

- Given a connected, weighted, undirected graph, find a spanning tree of minimum weight
- The above approaches suffice with minor changes:
  - Add nodes to partial tree approach: Prim's Algorithm
  - Add acyclic edges approach : Kruskal's algorithm

Prim's Algorithm

Given: a connected undirected weight graph

```
Initialize fringe to have a root node with costToTree = 0

all nodes are unvisited;

Repeat until all nodes are visited {
   Choose from fringe the unvisited node (n*) with minimum
costToTree;

  Add the corresponding edge to the spanning tree, set n* as visited

  for each (edge (n*, n') with one end-node n*) {
      if (n' is not visited) then add <n', (n*,n'), cost(n*,n')> into the fringe;
   }
}
```

Kruskal's Algorithm

Given: a connected undirected weight graph (*N* nodes, *M* edges)

```
Initialize an empty edge set T.

Sort all graph edges by the ascending order of their weight values.

For each edge in the sorted edge list
        Check whether it will create a cycle with the edges inside T.
        If the edge doesn't introduce any cycles, add it into T.
        If T has (V-1) edges, exit the loop.

return T
```

Complexity: Weighted Spanning Trees

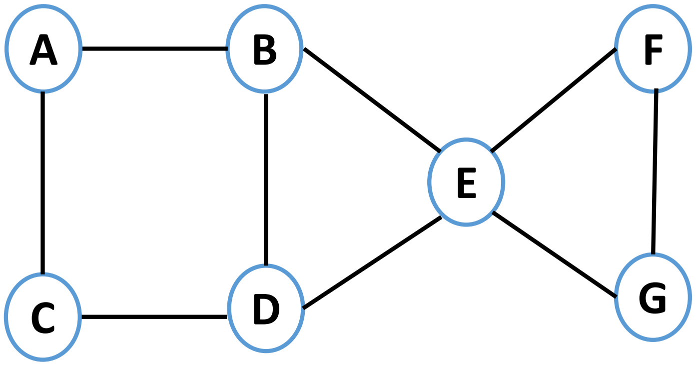
- Depends on data structure
  - Naïve approach, if using adjacency list with linear search
    - Prims : $\mathrm{O}(|\mathrm{V}^2|)$
    - Kruskal's: $\mathrm{O}(\text{sorting of edges} + |\mathrm{V}||V + \mathrm{E}|) \sim \mathrm{O}(|\mathrm{V}^2|)$

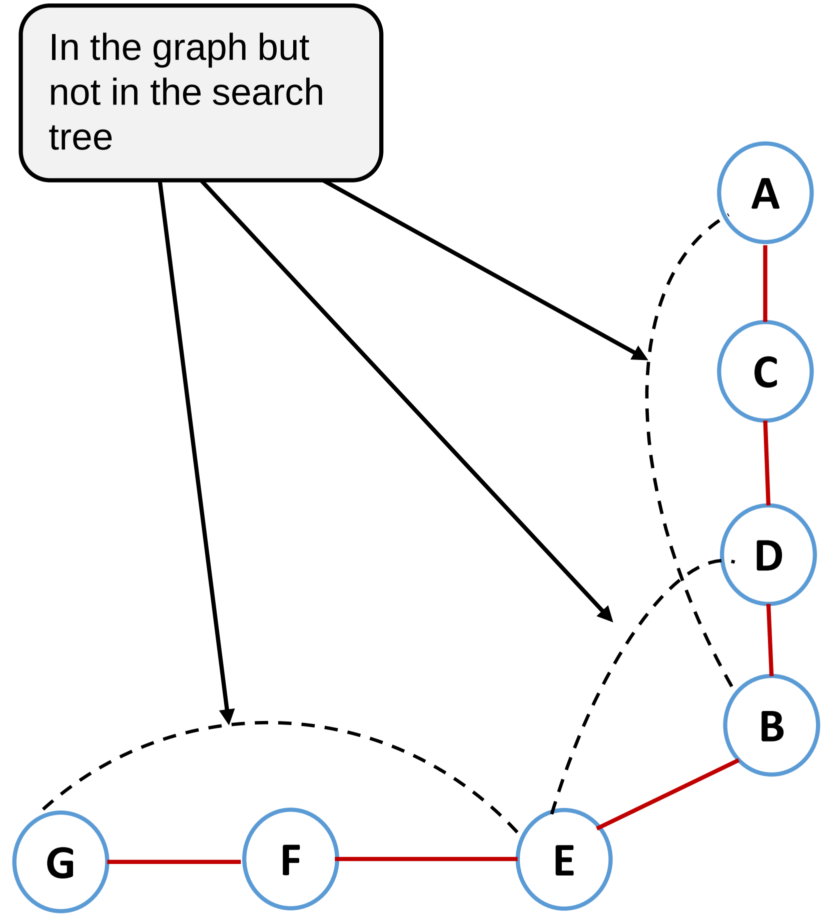
    // $|\mathrm{E}| \approx$ some multiple of $|\mathrm{V}|$

  - Priority queue
    - Prim's algorithm becomes similar to Dijkstra's
    - Complexity: $\mathrm{O}(|\mathrm{E}|\mathrm{Log}|\mathrm{V}|)$

- Can we do better in Kruskal's algorithm?
  - A new data structure: disjoint sets

Recap: Cycle Detection using DFS

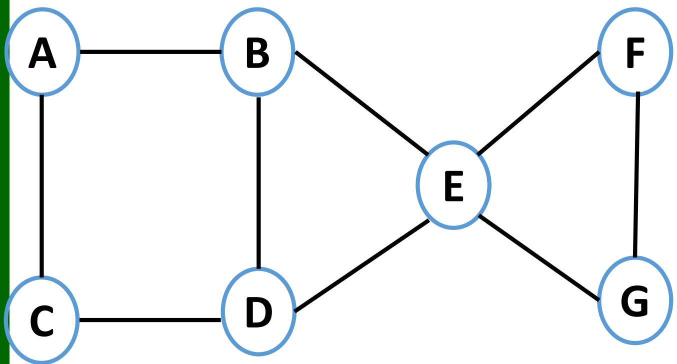




Back edge: Edge which is missing in the DFS tree but present in the graph

All the back edges which DFS skips over are part of cycles

Recap: Cycle Detection using DFS



- Steps:

- Start DFS traversal

- Keep track of parent of the node being visited

- If you find a node that has already been visited but is not the parent of the current node being visited – there is a cycle

0	A	C, B
1	B	A, D, E
2	C	A, D
3	D	C, B, E
4	E	B, D, F, G
5	F	E, G
6	G	E, F

DFS (A, -1), visited(A) = true

DFS (C, A), visited(C) = true

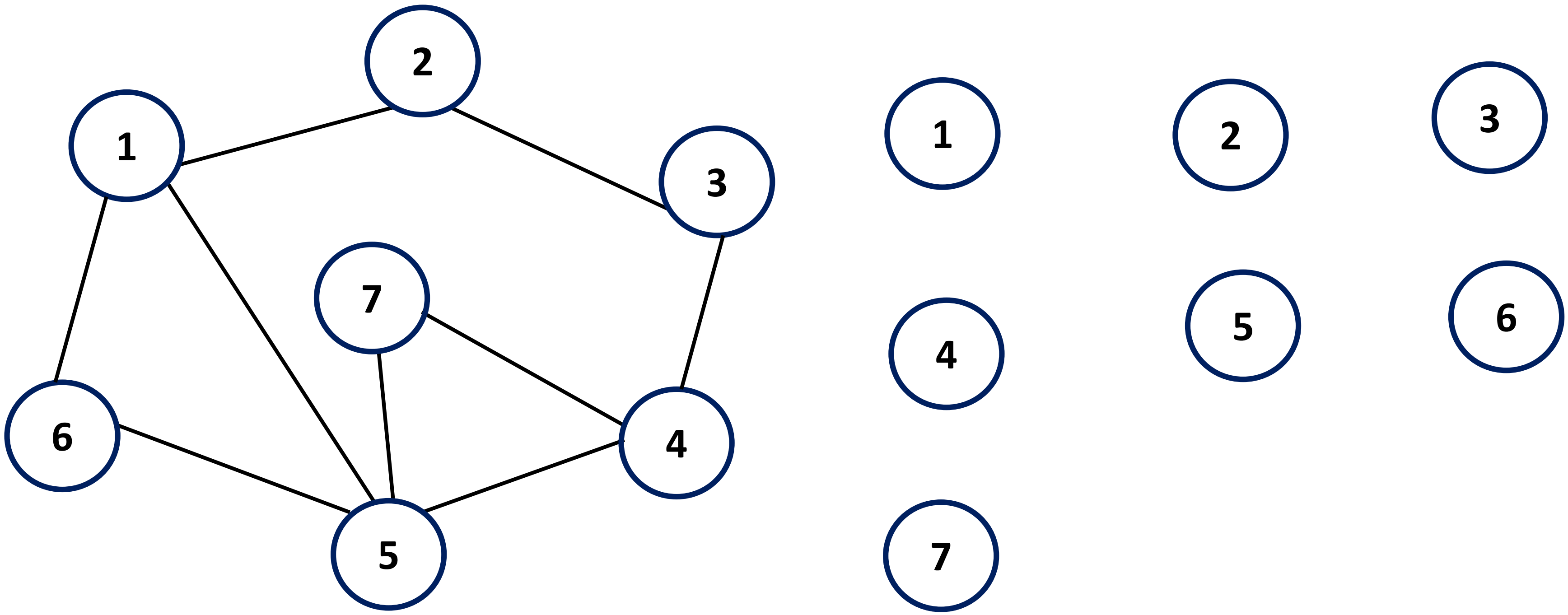
DFS (D, C), visited(D) =true

DFS (B, D), visited(B)=true

A has already been visited and A≠ parent(B). Cycle found

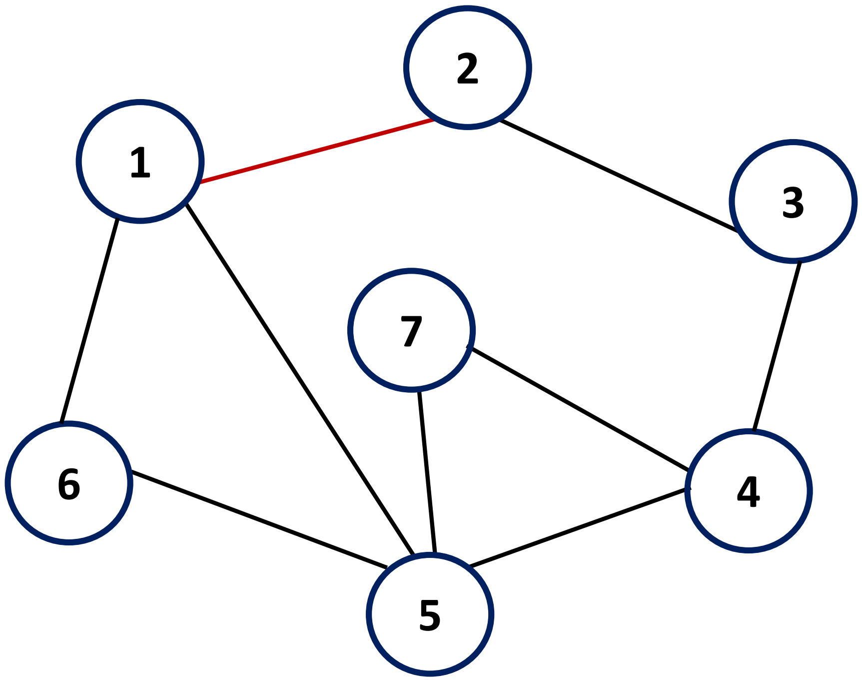
Finding a cycle – another approach

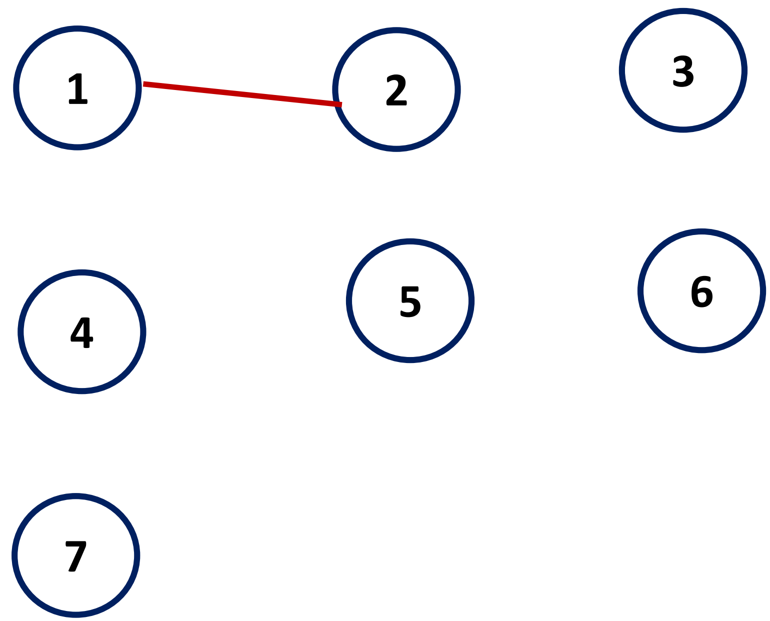
Order of edges considered: (1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,3), (4,7), (4,5)



Finding a cycle – another approach

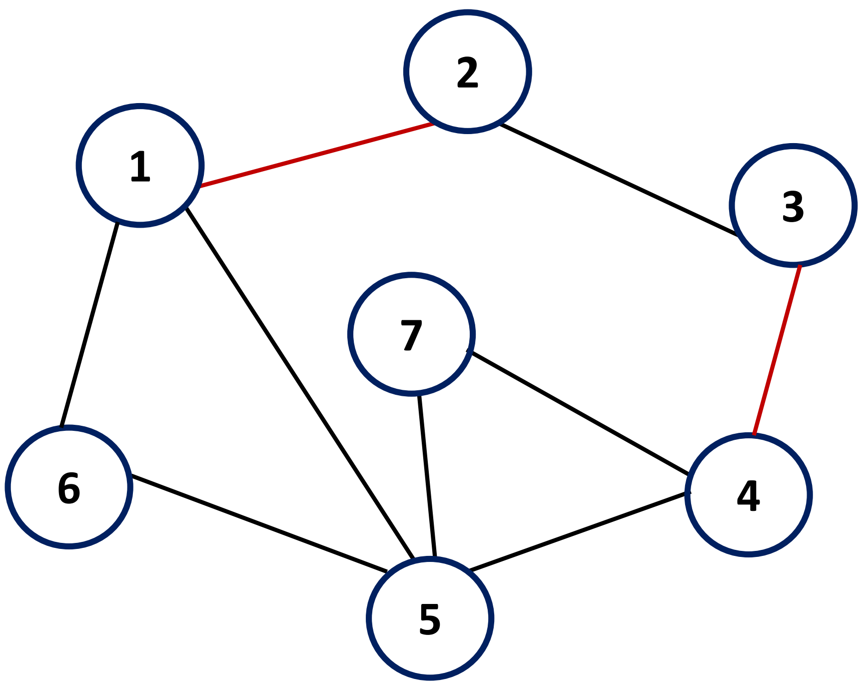
Order of edges considered: (1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,3), (4,7), (4,5)

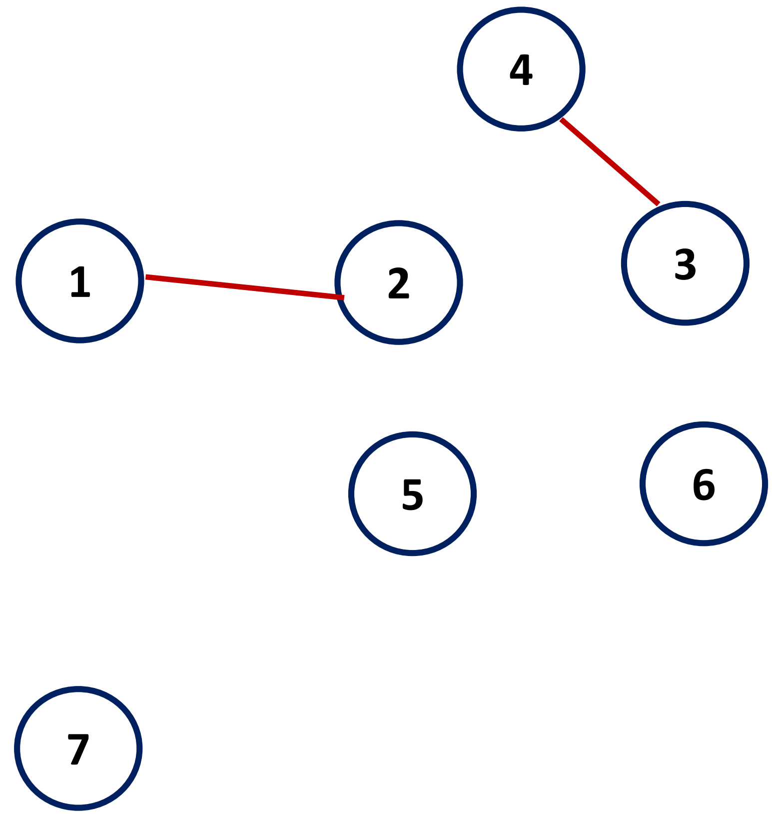




Finding a cycle – another approach

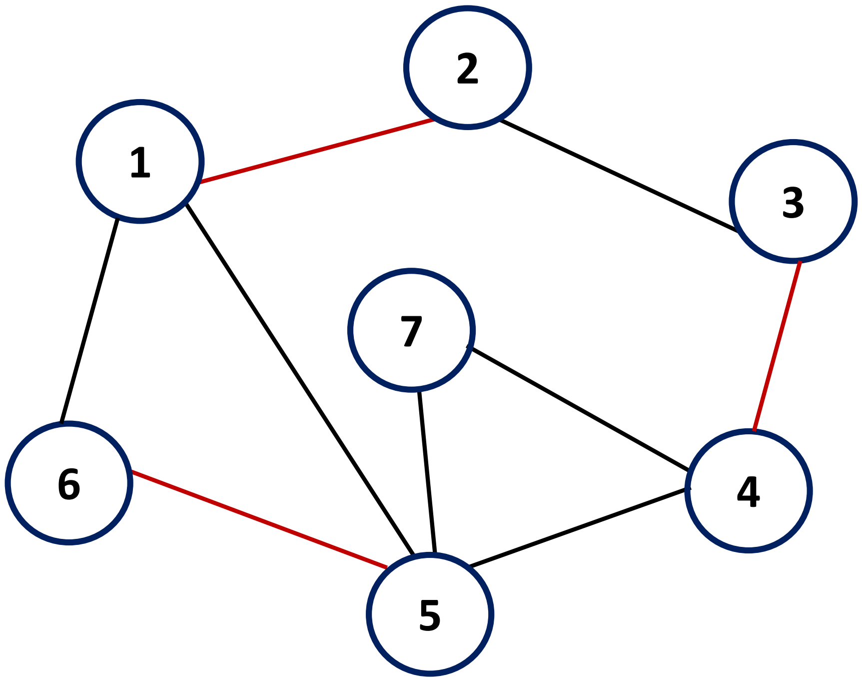
Order of edges considered: (1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,3), (4,7), (4,5)

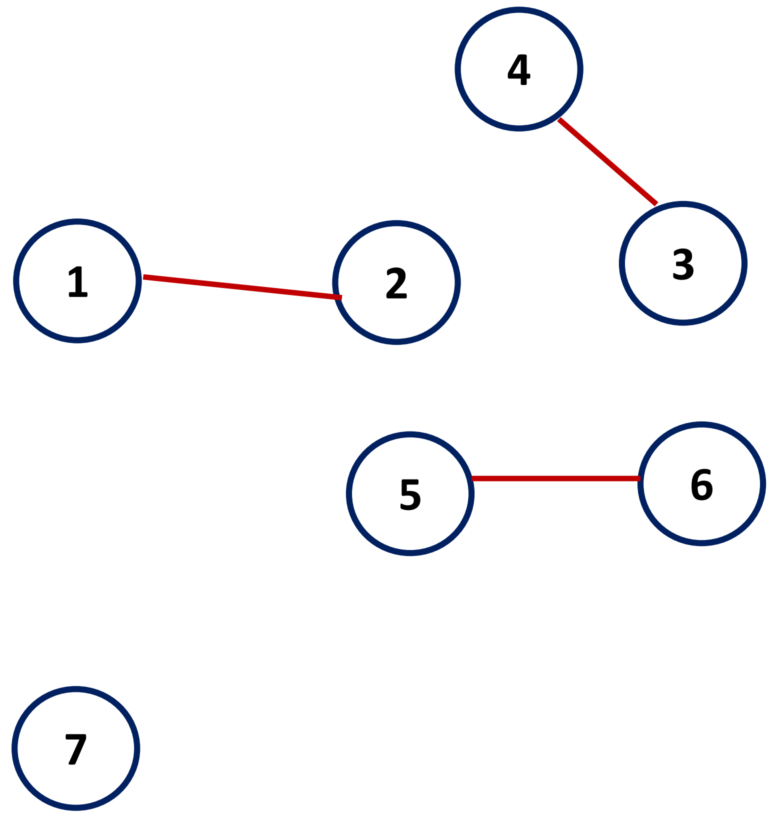




Finding a cycle – another approach

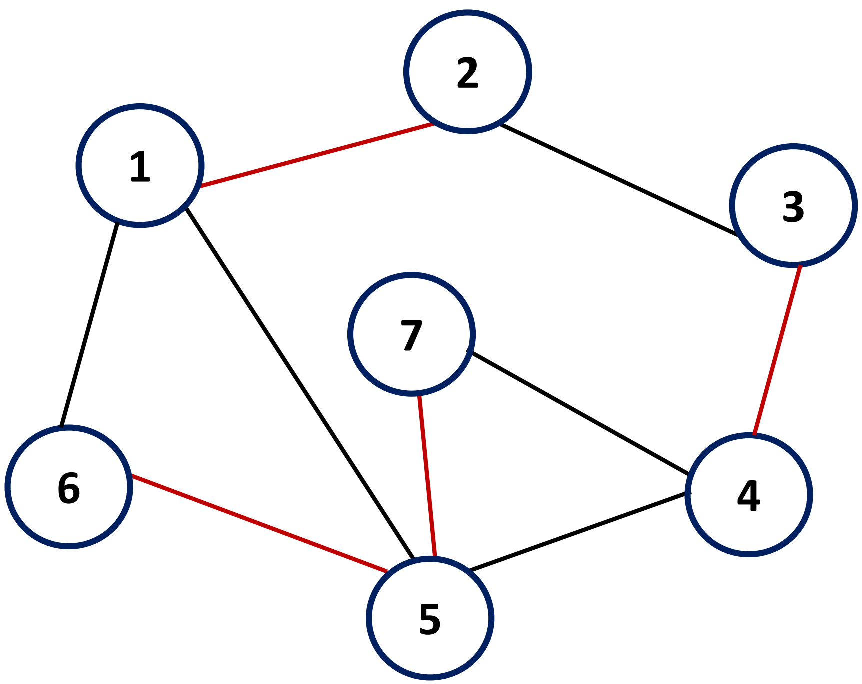
Order of edges considered: (1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,3), (4,7), (4,5)

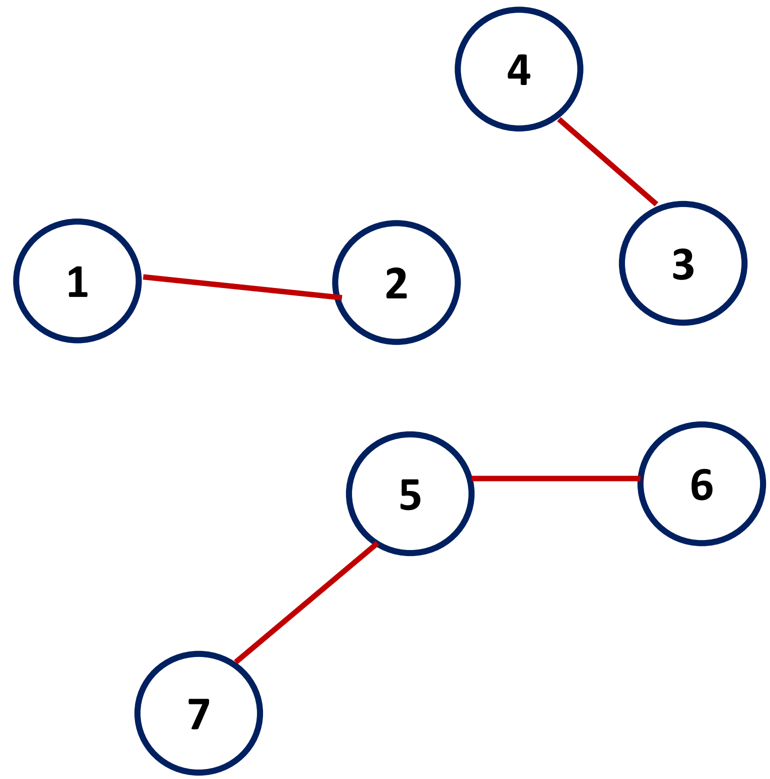




Finding a cycle – another approach

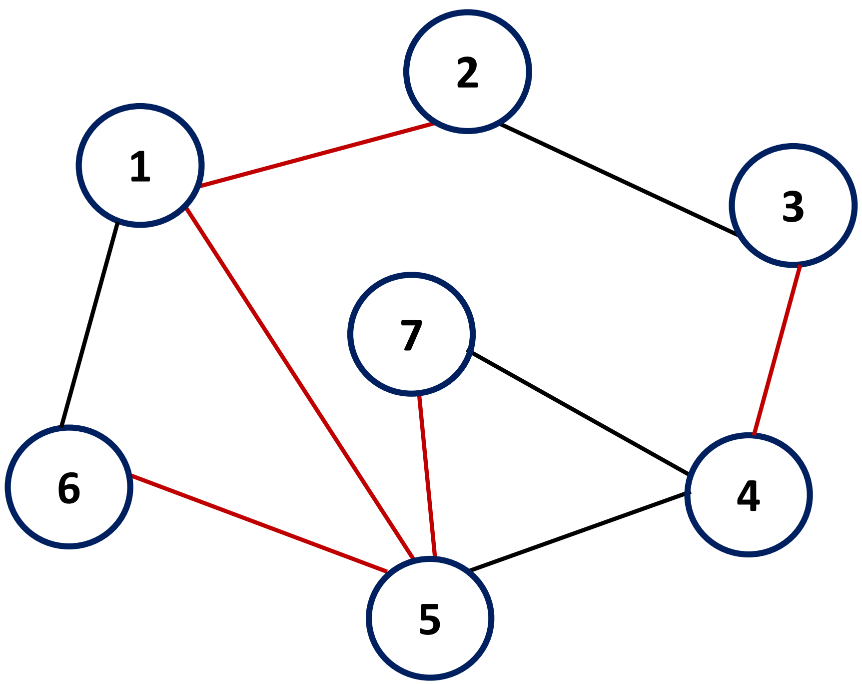
Order of edges considered: (1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,3), (4,7), (4,5)

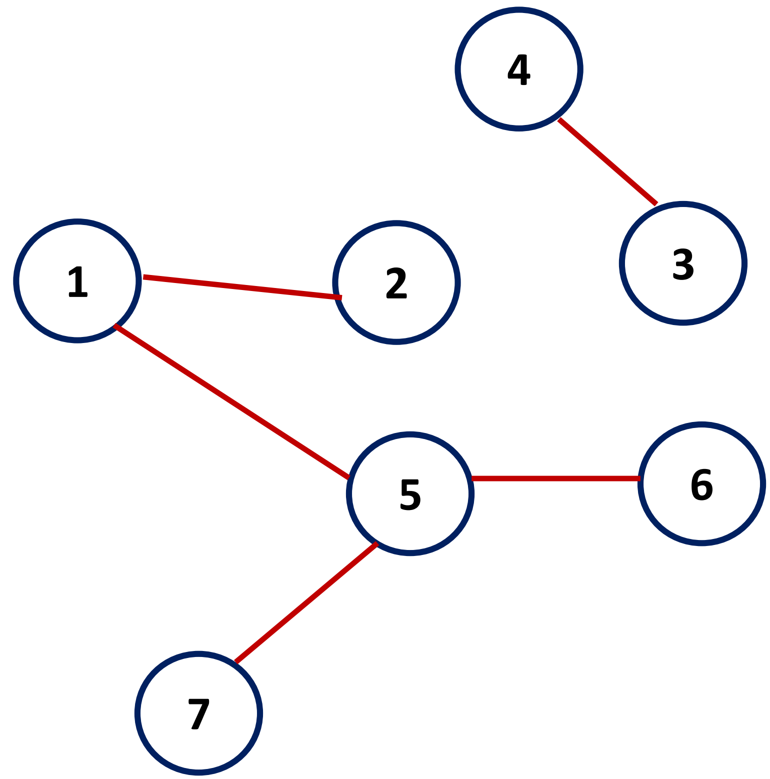




Finding a cycle – another approach

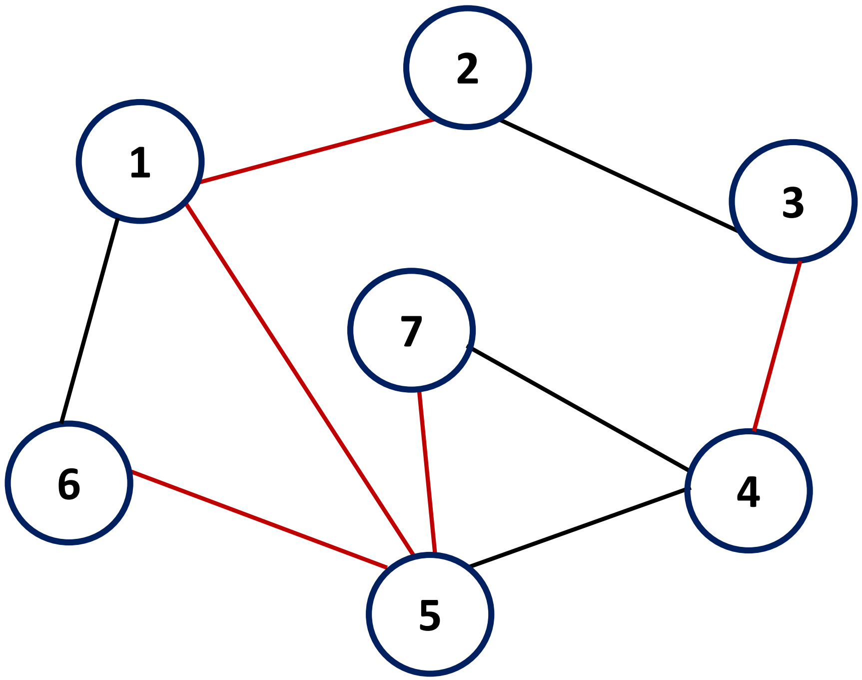
Order of edges considered: (1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,3), (4,7), (4,5)

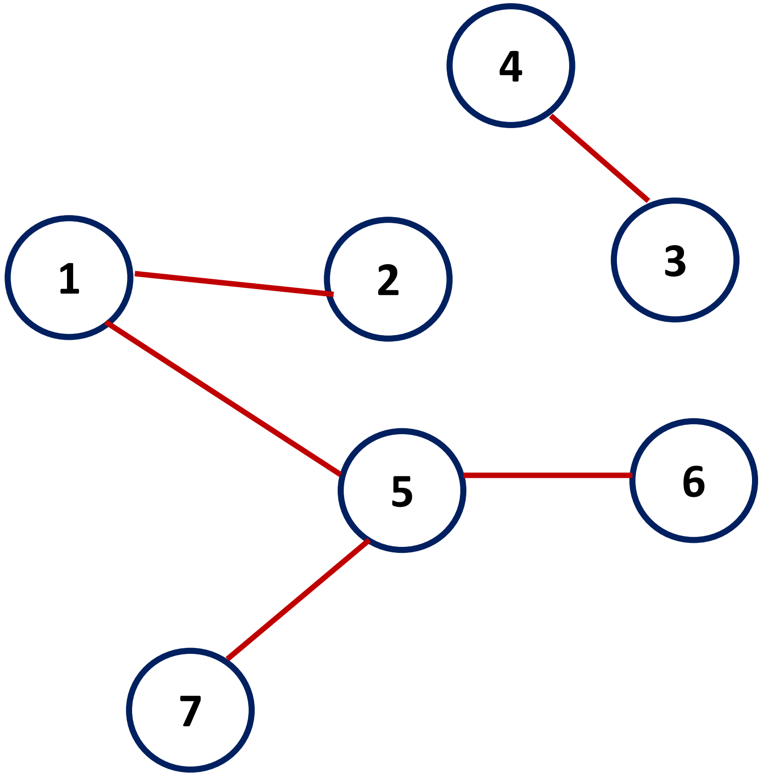




Finding a cycle – another approach

Order of edges considered: (1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,3), (4,7), (4,5)



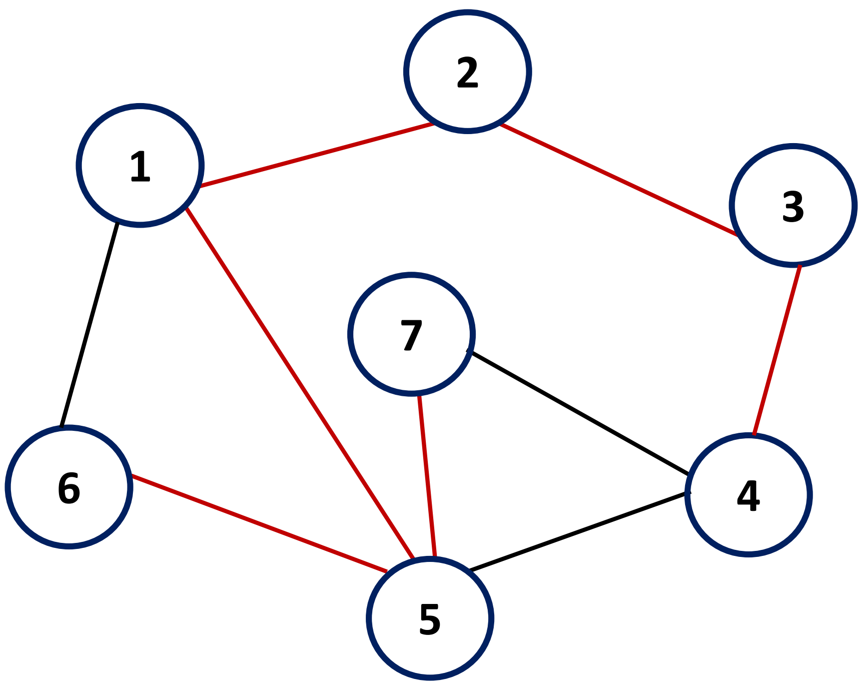


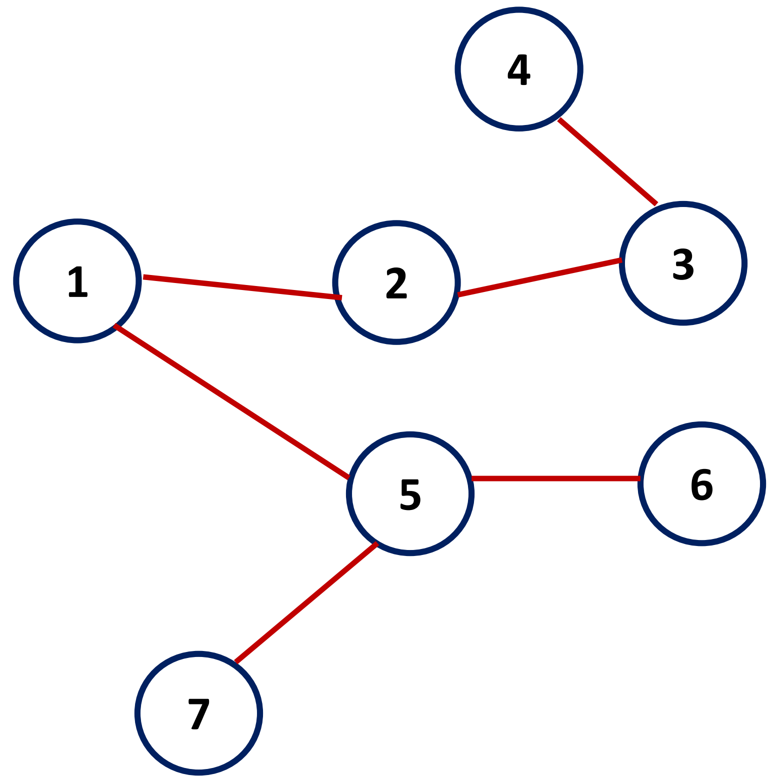
1 and 6 are in the same tree – ignore

Can Kruskal's use this approach?

Kruskal's with the new approach

Order of edges considered: (1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,3), (4,7), (4,5)

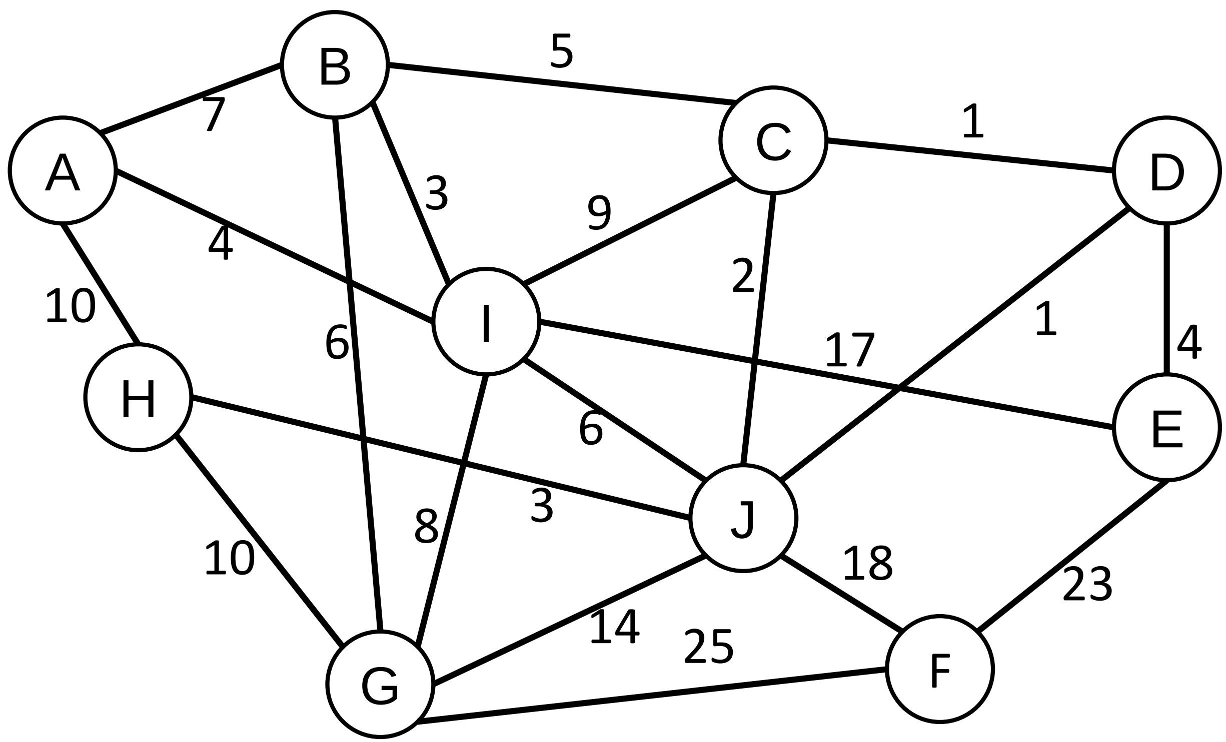




Stop when N-1 edges have been found

Kruskal's Algorithm

- Merge trees
  - Initially, each node is a single-node tree
  - At each step, merge two trees into one
  - The merge cost is the minimum (min-cost edge)



Kruskal's Algorithm

Given: a connected undirected weight graph (*N* nodes, *M* edges)

```
Set forest as N node sets, each containing a node;

Set fringe as a priority queue of all the edges ⟨n1, n2, length⟩;

Set tree as an empty set of edges;

Repeat until forest contains only one tree or edges is empty {
    Get and remove ⟨n1*, n2*, length*⟩ as the edge with minimum length from
fringe;
    If (n1* and n2* are in different sets in forest) {
        Merge the two sets in forest;
        Add the edge to tree;
    }
}
return tree;
```

Kruskal's Algorithm

Given: a connected undirected weight graph (*N* nodes, *M* edges)

Set forest as *N* node sets, each containing a node;

Set fringe as a priority queue of all the edges ⟨n1, n2, length⟩;

Set tree as an empty set of edges;

Repeat until forest contains only one tree or edges is empty {
Get and remove ⟨n1*, n2*, length*⟩ as the edge with minimum length from fringe;
If (n1* and n2* are in different sets in forest) {
Merge the two sets in forest;
Add the edge to tree;
}
}
return tree;

Need a way to –

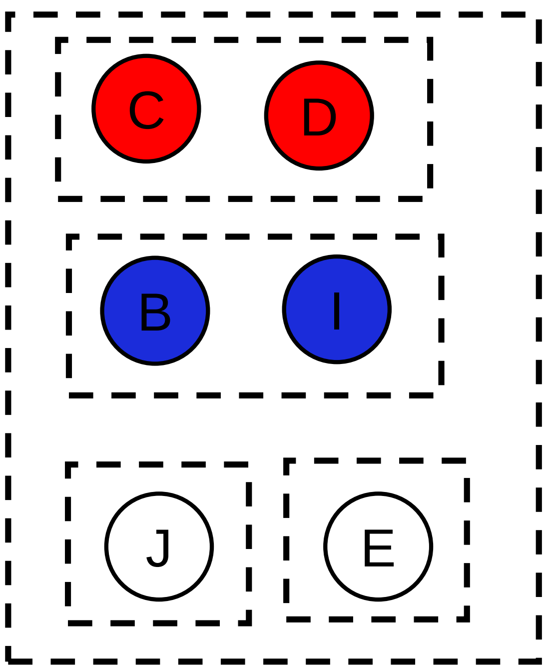
- Efficiently **find** if two nodes are in sets in a forest
- Efficiently merge (**Union**) two trees

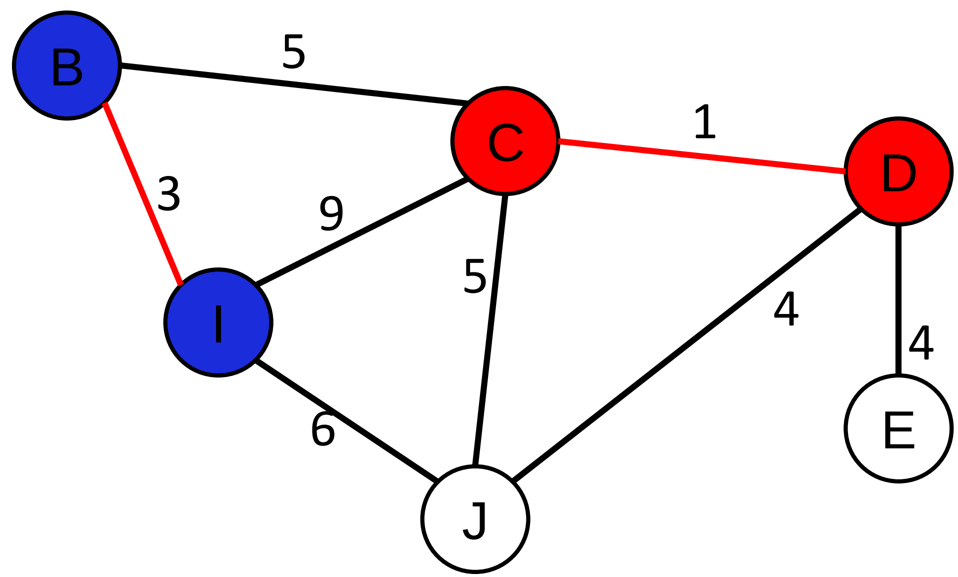
A new data structure: disjoint sets

Find and Union Operation

- **Find**: Determine whether two elements belong to the same set
- **Union**: merge two sets into one
- The cost of Find and Union depends on the data structure of the forest: set of sets

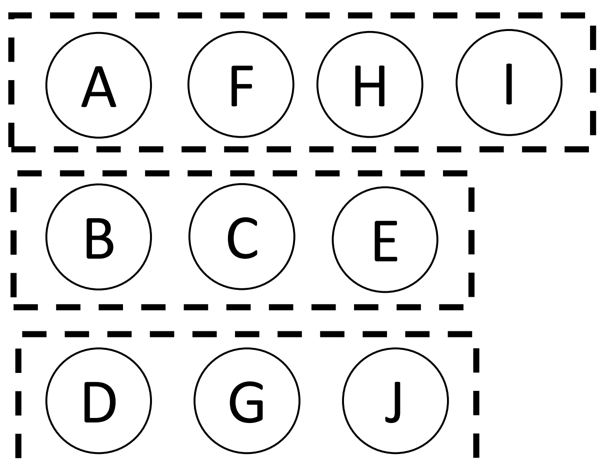
forest





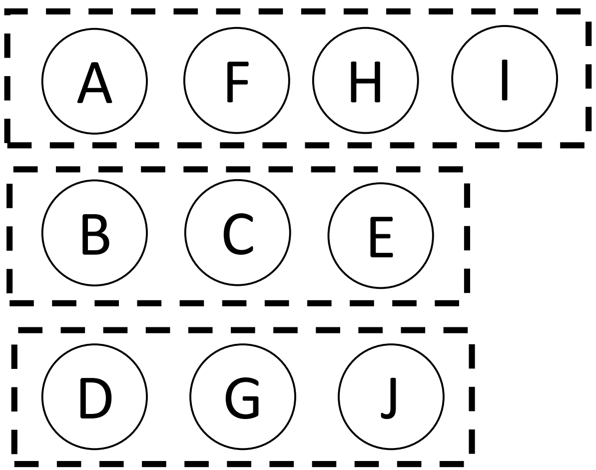
Set of Sets: Data Structures

- Option 1: set of sets (e.g. `HashSet<HashSet<Node>>`)
  - Cost of find: iterate over all sets
  - Cost of union: add all the elements from one set to another



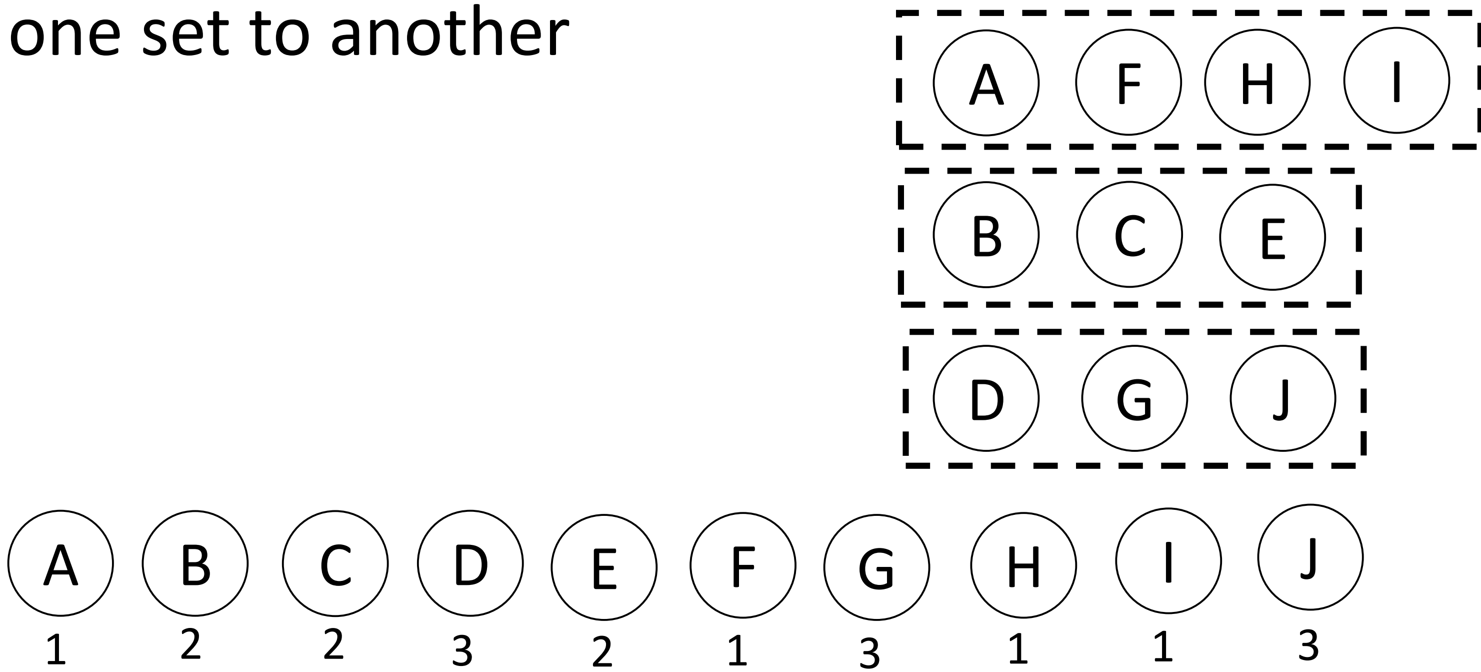
Set of Sets: Data Structures

- Option 1: set of sets (e.g. `HashSet<HashSet<Node>>`)
  - Cost of find: iterate over all sets, $O(n)$
  - Cost of union: add all the elements from one set to another, $O(n)$



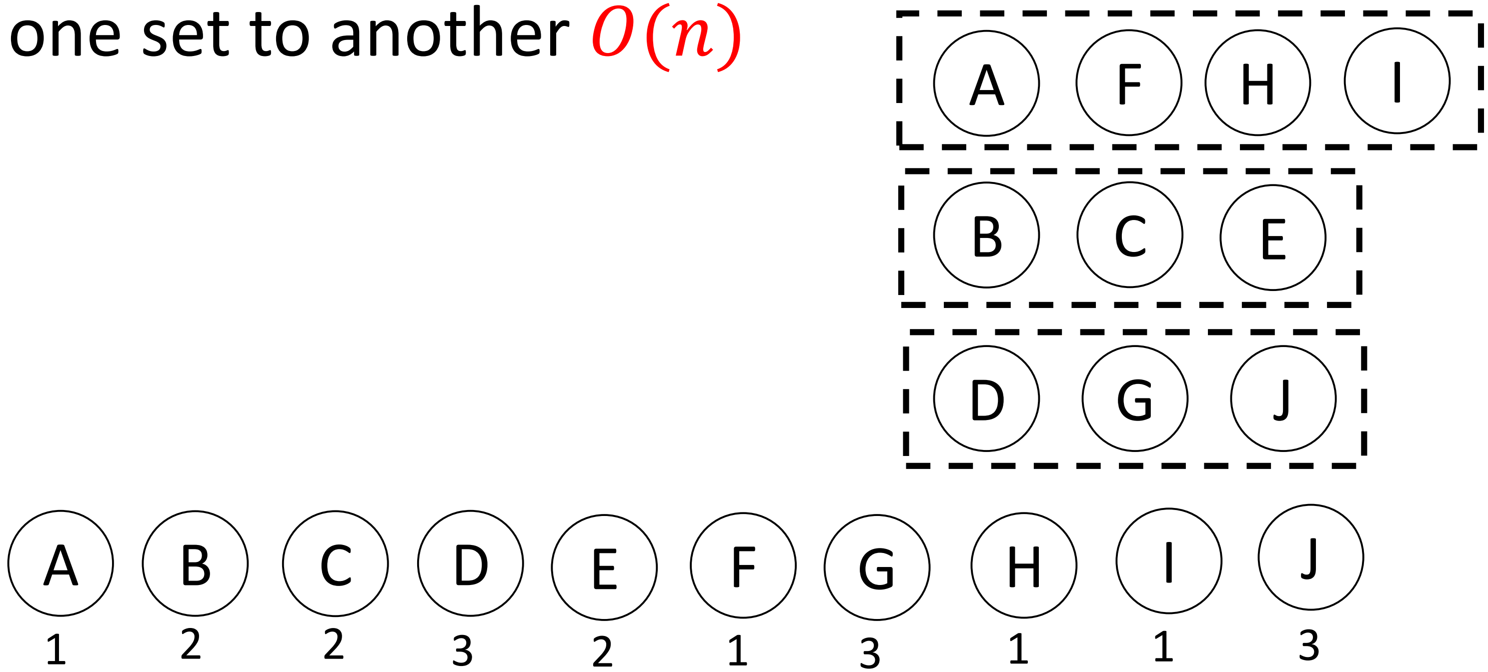
Set of Sets: Data Structures

- Option 2: mark each node with set ID
  - Cost of find: check whether the two elements have the same set ID
  - Cost of union: iterate all the nodes, change the set ID of one set to another



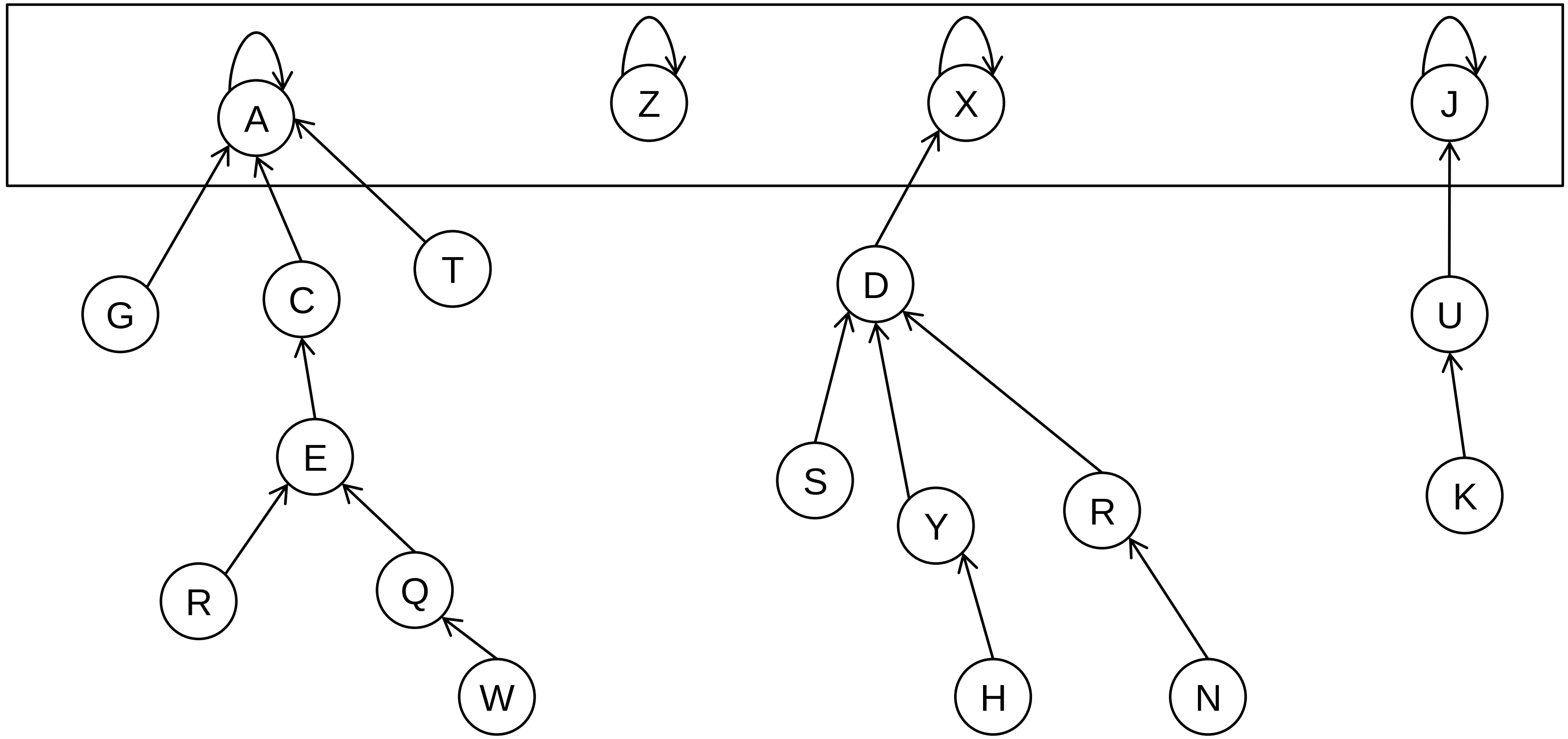
Set of Sets: Data Structures

- Option 2: mark each node with set ID
  - Cost of find: check whether the two elements have the same set ID $O(1)$
  - Cost of union: iterate all the nodes, change the set ID of one set to another $O(n)$



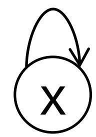
Set of Sets: Data Structures

- Option 3 (the best): disjoint-set (union-find) data structure
  - Set of inverted trees
  - Each set is represented by a linked tree with links pointing **towards** the root
  - **Forest** = set of root nodes

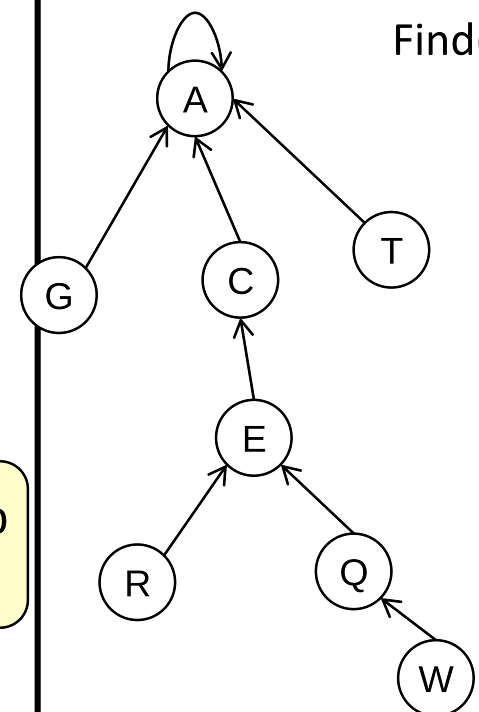


Disjoint Set

```
// make a new set with element x
MakeSet(x) {
  x.parent = x;
  add x to forest;
}
// find the root of the set that x belongs to
Find(x) {
  if (x.parent == x) { // x is the root
    return x;
  } else {
    root = Find(x.parent);
    return root;
  }
}
```



Recursively go up to the root

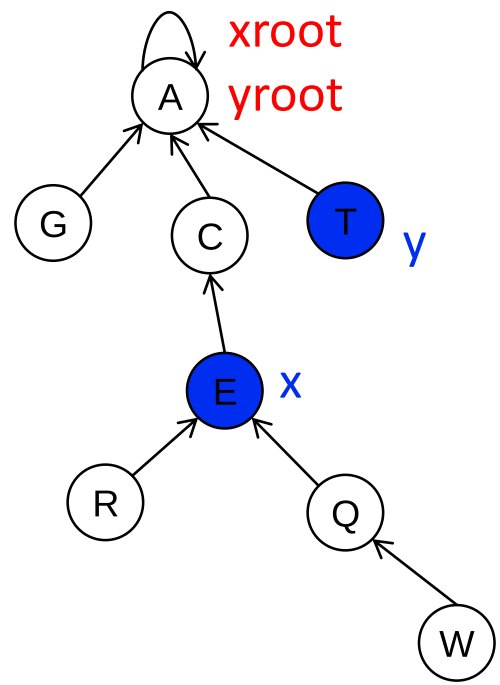


Find(A) = A
Find(G) = A
Find(E) = A
Find(W) = A
...

Disjoint Set

```
// union the sets of x and y
Union(x, y) {
   xroot = Find(x);
   yroot = Find(y);
   if (xroot == yroot) {
      // x and y belong to
      // the same set
      return;
   } else {
      xroot.parent = yroot;
      remove xroot from forest;
   }
}
```

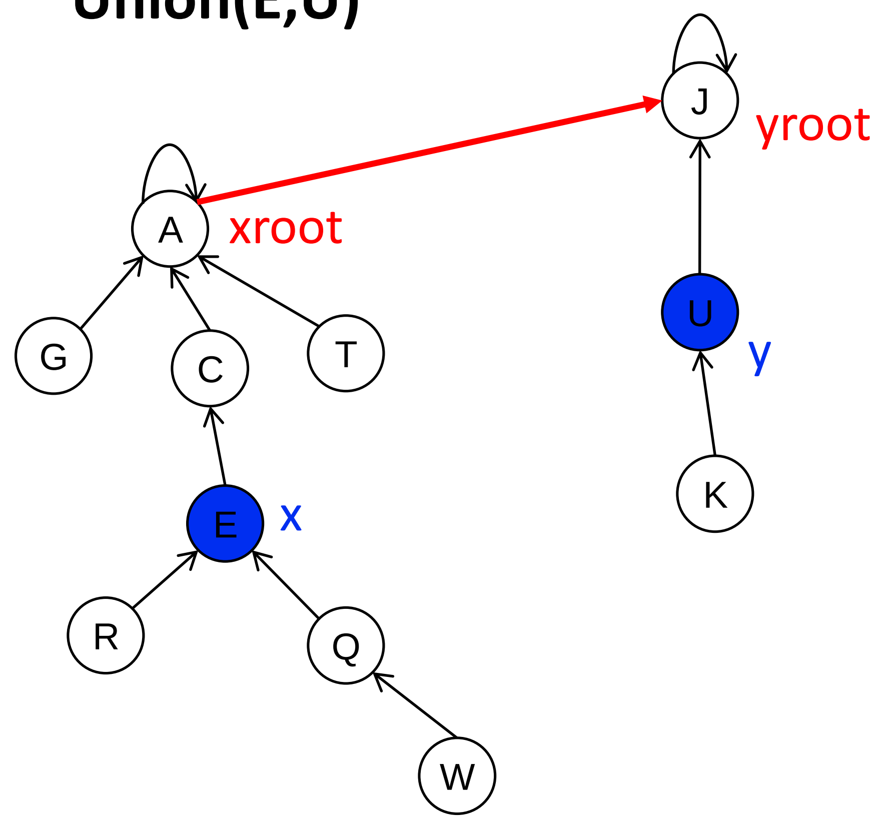
Union(E,T)



Disjoint Set

```
// union the sets of x and y
Union(x, y) {
   xroot = Find(x);
   yroot = Find(y);
   if (xroot == yroot) {
      // x and y belong to
      // the same set
      return;
   } else {
      xroot.parent = yroot;
      remove xroot from forest;
   }
}
```

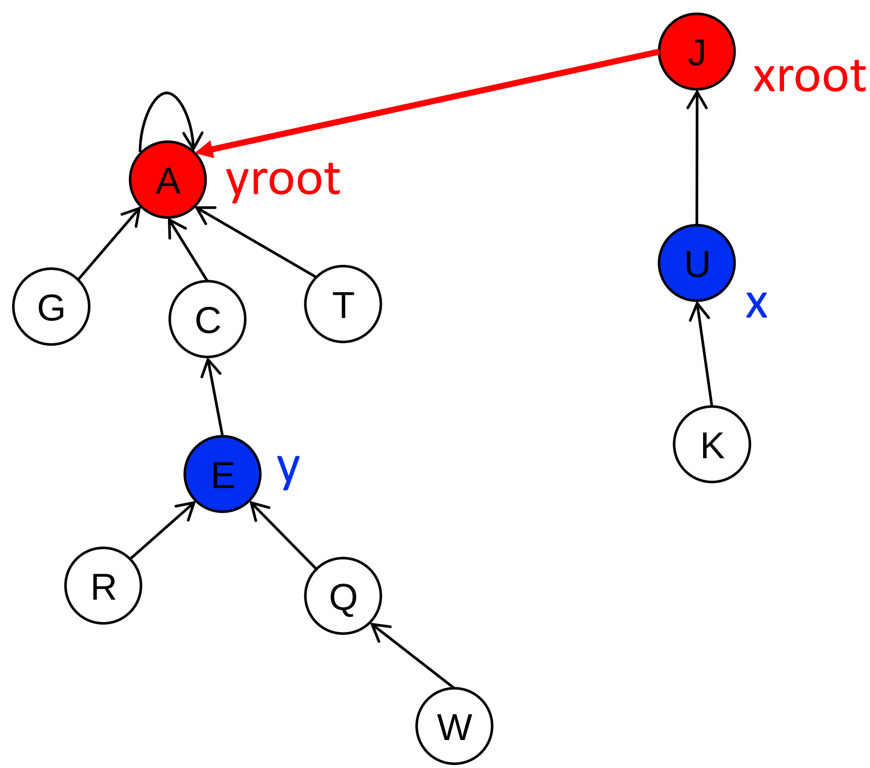
Union(E,U)



Disjoint Set

```
// union the sets of x and y
Union(x, y) {
  xroot = Find(x);
  yroot = Find(y);
  if (xroot == yroot) {
    // x and y belong to
    // the same set
    return;
  } else {
    xroot.parent = yroot;
    remove xroot from forest;
  }
}
```

Union(U,E)



Order can change the depth of the resultant tree

Disjoint Set

- To reduce complexity, always merge shorter trees into deeper ones

```
MakeSet(x) {
    x.parent = x;
    x.depth = 0;
    add x to forest;
}

Find(x) {
    if (x.parent == x) {
        return x;
    } else {
        root = Find(x.parent);
        return root;
    }
}
```

```
Union(x, y) {
    xroot = Find(x);
    yroot = Find(y);
    if (xroot == yroot) {
        return;
    } else {
        if (xroot.depth < yroot.depth) {
            xroot.parent = yroot;
            remove xroot from forest;
        } else {
            yroot.parent = xroot;
            remove yroot from forest;
            if (xroot.depth == yroot.depth)
                xroot.depth ++;
        }
}
```

Kruskal's with Disjoint Sets

- If we use Disjoint sets data structure.
  - We maintain each connected component as a disjoint set:
  - Initially each vertex is in its own disjoint set: $O(|V|)$
  - Edges are maintained in a sorted order or in a priority queue: $O(|E| \log |E|)$
  - When considering an edge (u, v) to include, we check if u and v are in the same disjoint set (Find operations):
    - If yes, they form a cycle
    - Else, we include the edge and perform a Union operation on disjoint sets containing u and v.
    - Since we always merge shorter trees into deeper ones, the worst case time complexity of Find and Union is $O(\log|V|)$
    - In worst case we iterate through all edges: $O(|E| \log|V|)$.
    - Overall complexity: $O(|V| + (|E| \log|E|) + (|E| \log|V|) = O(|E| \log|E|)$ (as $|E| >= |V| - 1$)

Test: May 9

Syllabus: Everything we've covered since week 7 up to today

Best of luck for the test!