Data Structures and Algorithms

XMUT-COMP 103 - 2026 T1

Graph

Agatha Rachmat

School of Engineering and Computer Science

Victoria University of Wellington

Topics

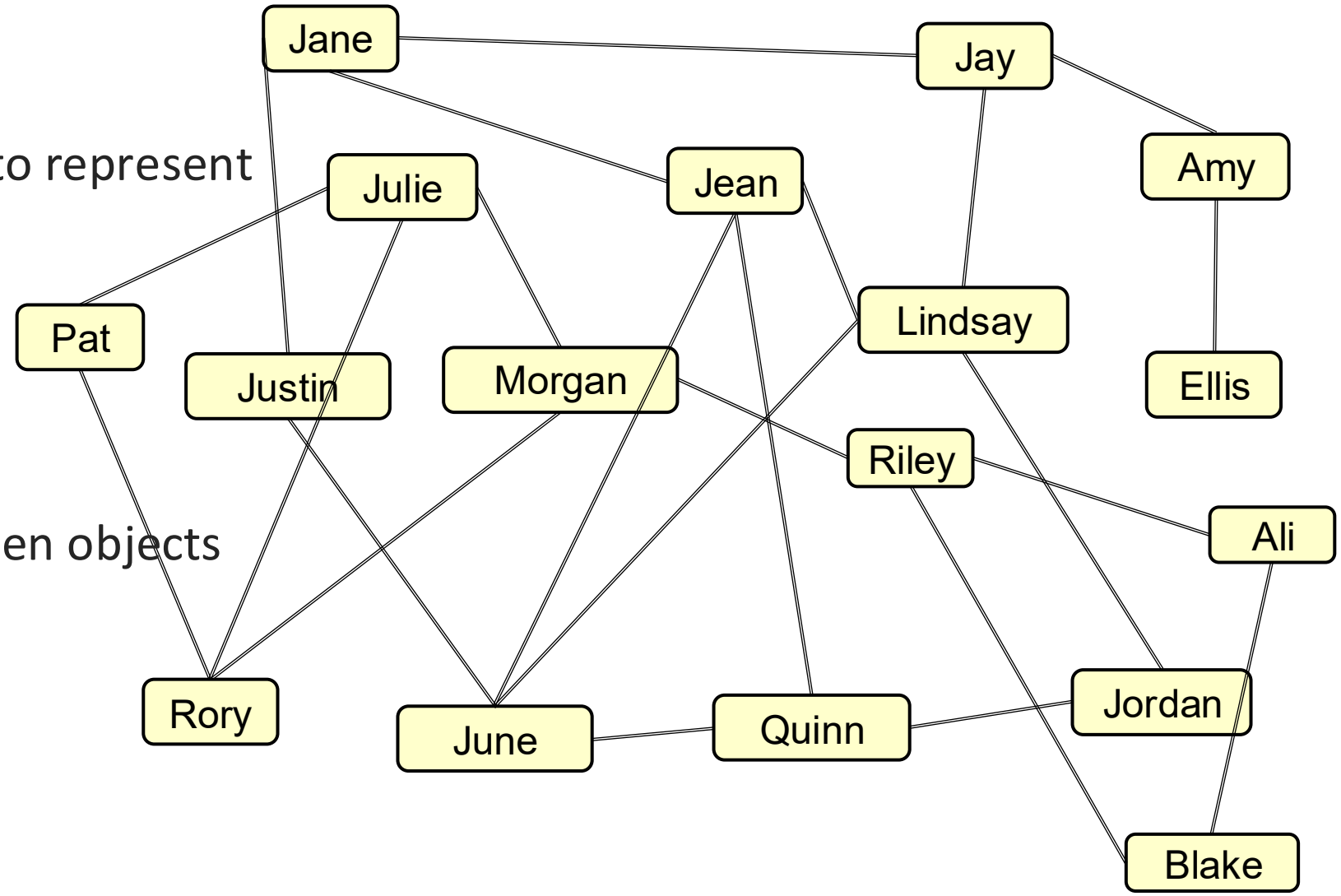
- Introduction to graph.
- Comparison of graph vs. tree.
- Example of graphs in real life.
- Programming with graph:
  - Traversing graph.
  - Improved traversing graph with marked node:
    - Set visited node.
    - Set visited flag.
  - Count connected nodes.
  - Connected to.

Graphs

A **graph** is a data structure used to represent connections between objects.

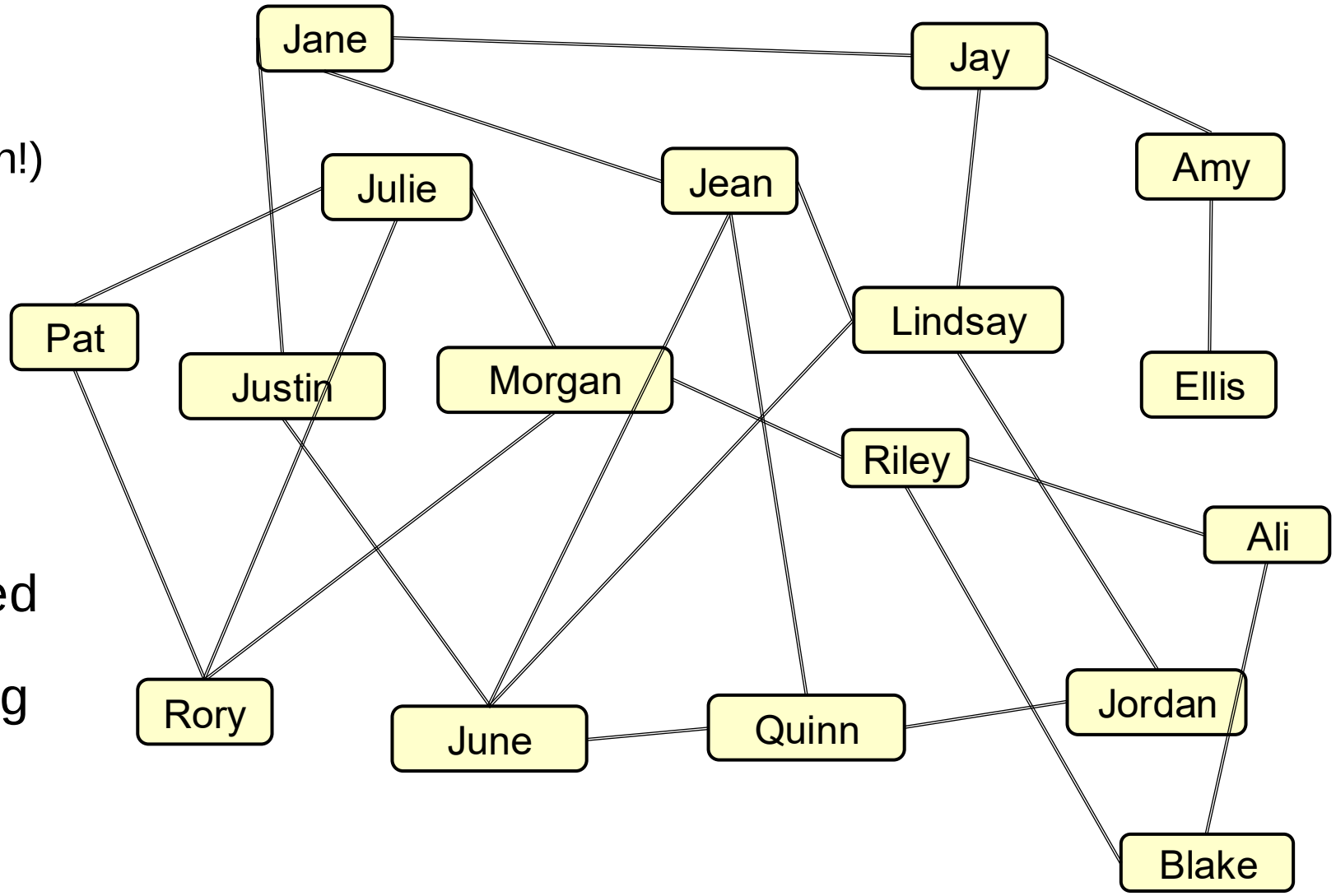
A graph consists of:

- **Vertices (nodes)** → the objects
- **Edges** → the connections between objects



Graphs

- Graphs are like trees:
  Nodes and Links (edges)
  (Trees are a special kind of graph!)
- Nodes have neighbours
  rather than children
- Graphs don't have a "root"
  (typically)
- Graphs may not be connected
- Can traverse a graph, starting at node, but
- **Graphs have cycles!**
- Lots of varieties of graphs – this one is the simplest.

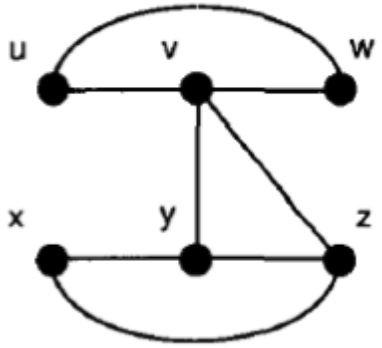


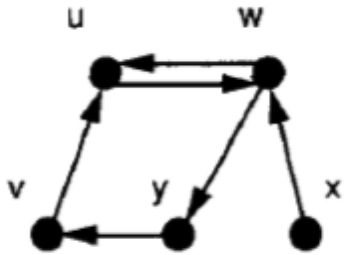
Definition

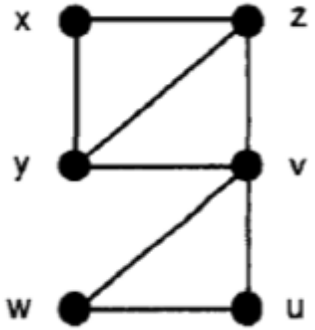
- A graph is a set of objects, called nodes/vertices (V), together with a collection of pairwise connections between them, called links/edges (E).

$$G = (V, E)$$

- Edge connections:
  - Directed graph: An edge (u,v) is directed from u to v if the pair is ordered, with u preceding v.
  - Undirected graph: the pair (u,v) is not ordered.
  - Mixed graph: contain both directed and unidirected graphs.



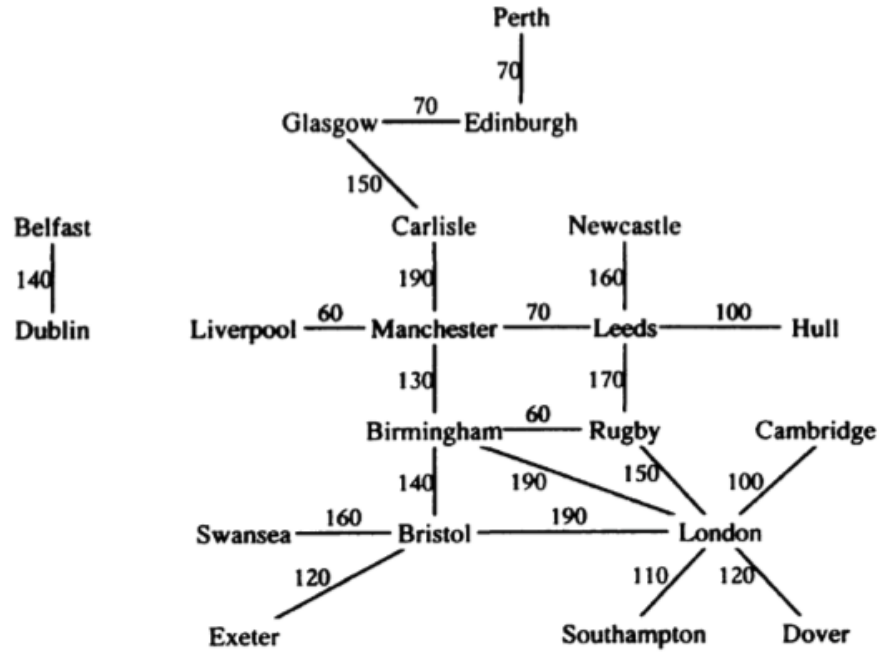


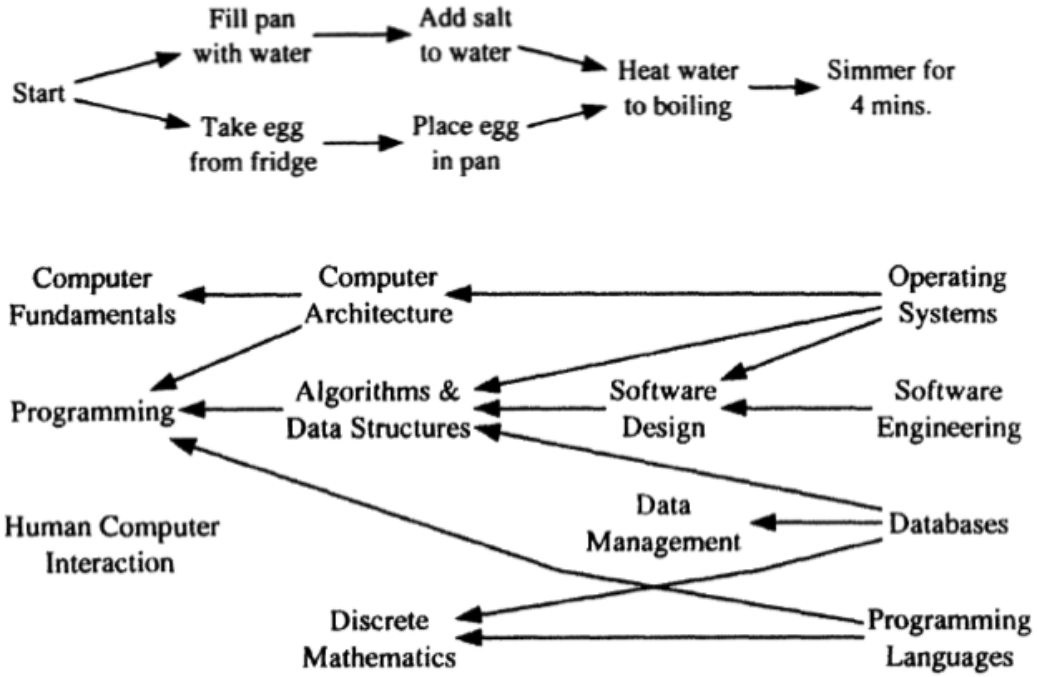


- Nodes and links, (or vertices and edges, if you are a mathematician)

Mechanics of Graph

- A path in a graph is a list of nodes such that each node and its successor in the list are connected by an edge in the graph.
- A cycle in a graph is a path whose first and last nodes are the same.
- An acyclic graph is one in which there is no cycle.)



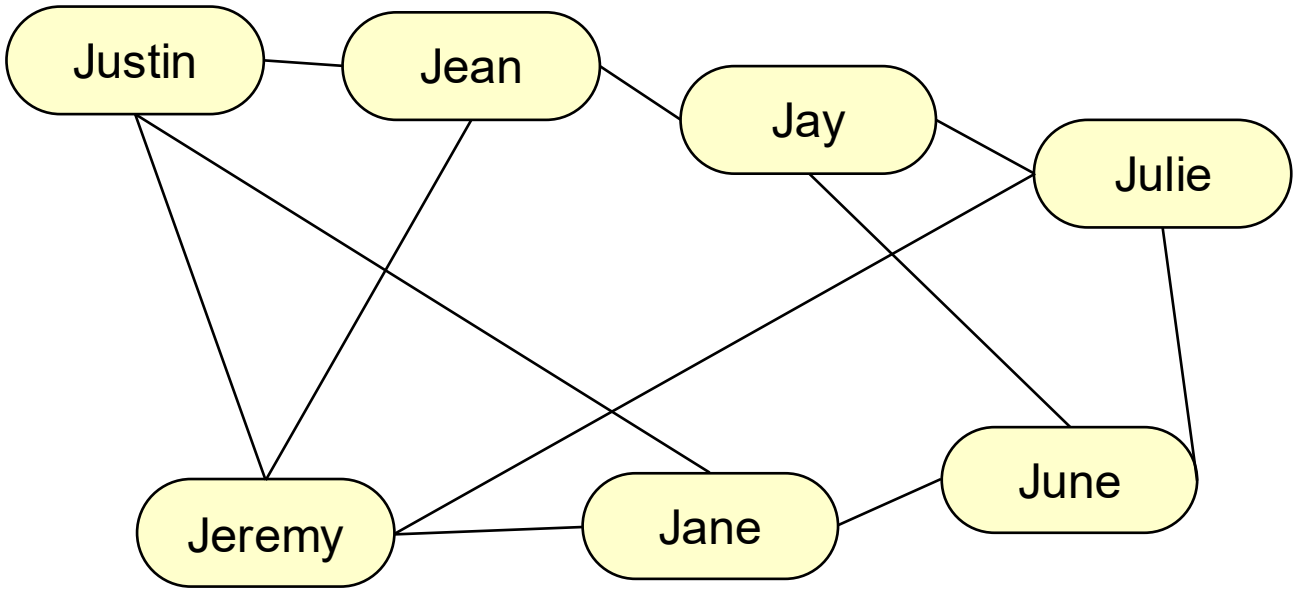


Comparison of Graph vs Tree

CRITERIA	TREE	GRAPH
Path	Only one between two vertices.	More than one path is allowed.
Root node	It has exactly one root node.	Graph doesn't have a root node.
Loops	No loops are permitted.	Graph can have loops.
Complexity	Less complex	More complex comparatively
Traversal techniques	Pre-order, In-order and Post-order.	Breadth-first search and depth-first search.
Number of edges	n-1 (where n is the number of nodes)	Not defined
Model type	Hierarchical	Network

Graph/Network Structured Data

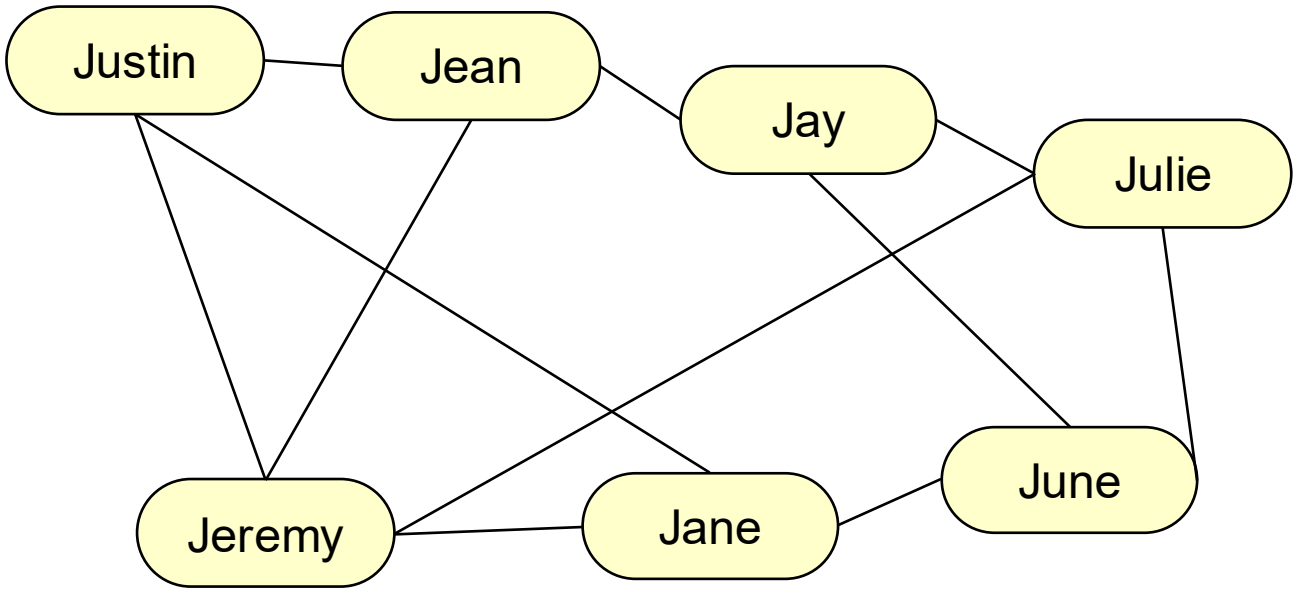
- Examples:
  - Social networks
  - Circuit diagrams
  - Network structures (communication,airline,…)
  - Road maps,
  - Database structure diagrams
  - whenever there are relationshipsbetween data items.



Graph/Network Structured Data

Social networks

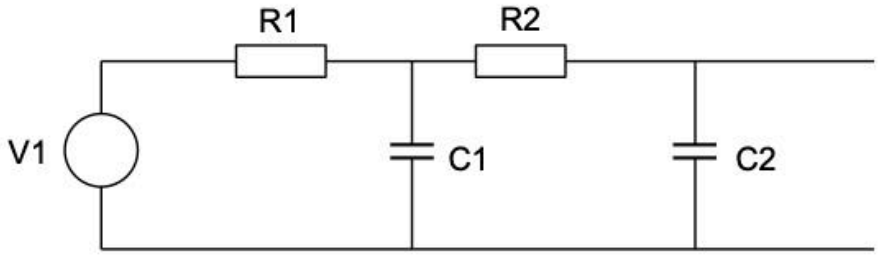
- Links between people (aka. friendship) - formed between two friends
- Symmetric relationship – if A is a friend of B, then B is also a friend of A.
- Cycle of friendships between people is limitless (i.e. has no end).

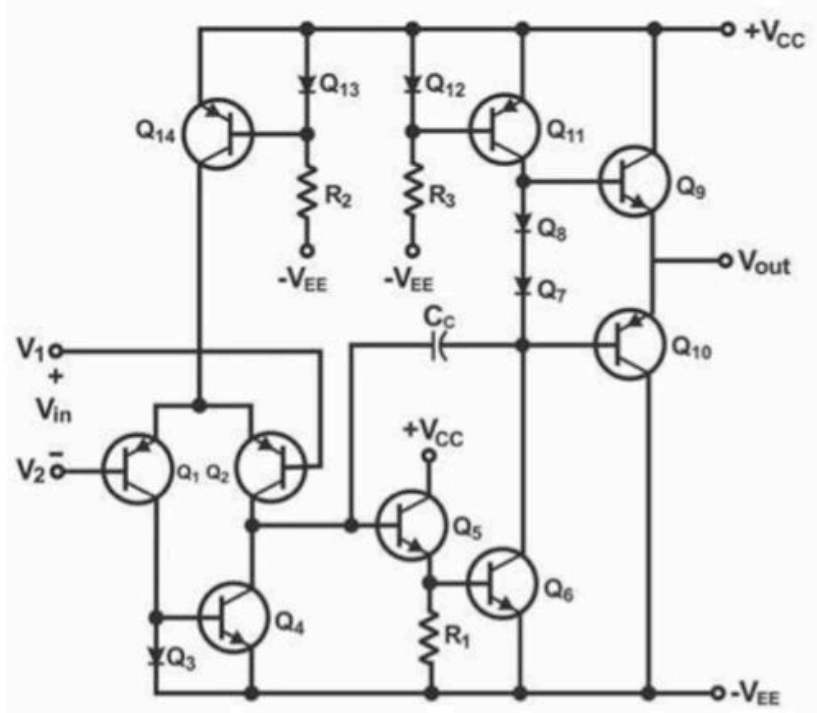


Graph/Network Structured Data

Circuit diagram:

- Interconnection of electrical components.
- In the circuit diagram, nodes are the electrical components and links are the electrical nodes.
- There must be return path for a closed loop for current to flow.
- Must have at least one source of energy.
- Might contain active and passive components.
- Direction of flow of the current is from positive to negative terminal of the source.
- One of the link is the reference (ground).





Graph/Network Structured Data

Network structures (communication,airline,…):

- Vertices are associated with airports.
- Edges are associated with flights.
- Two vertices are adjacent if there is at least one direct edge between them.
- Outgoing edges correspond to outbound flights from the given airport.
- Ingoing edges correspond to inbound flights to the given airport.
- Network is directive graph.

Graph/Network Structured Data

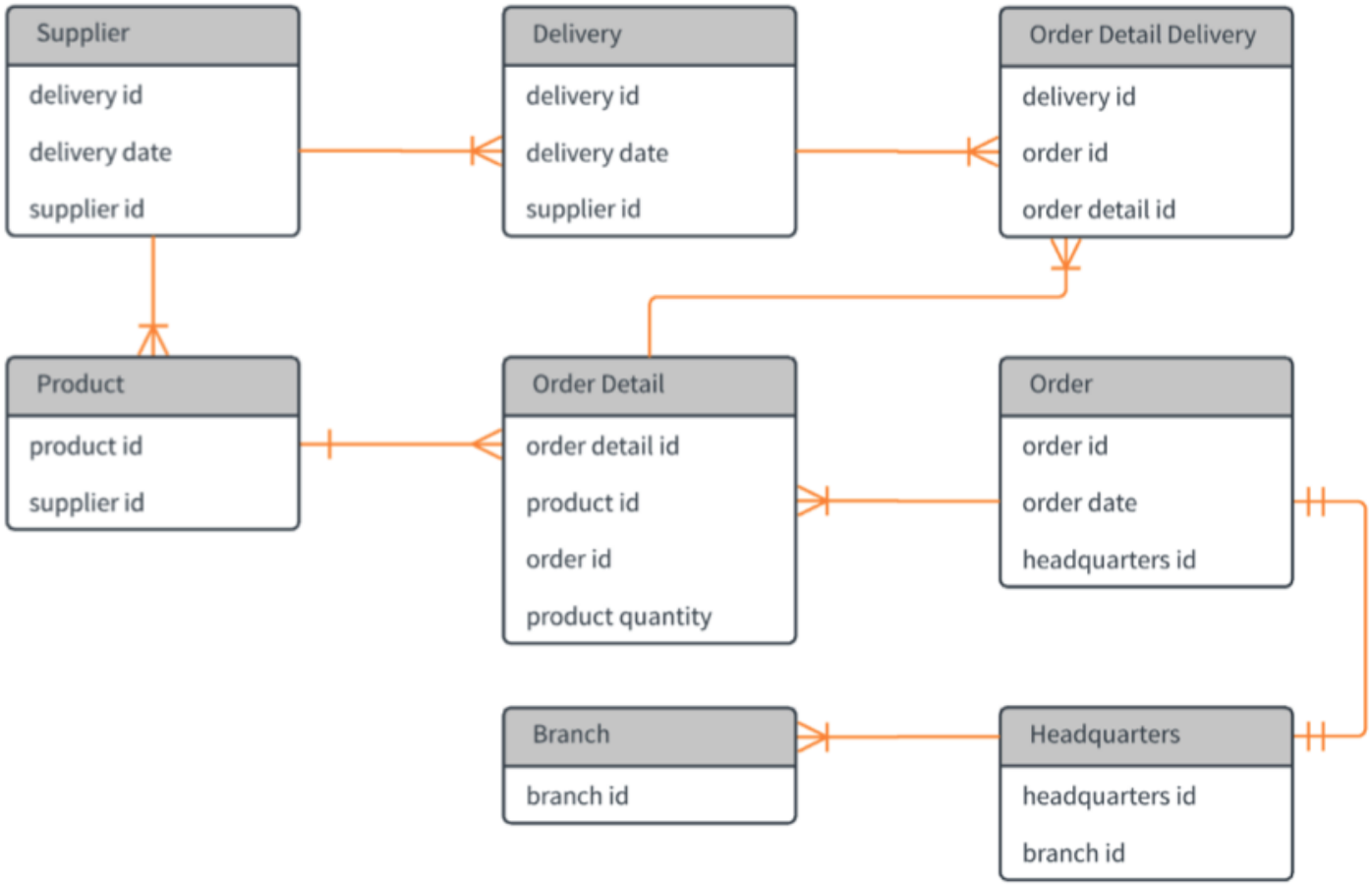
Road maps:

- Very similar to the airlines graph, but it is different.
- Vertices are associated with city.
- Edges are associated with road connecting the cities.
- Two vertices are adjacent if there is at least one road connecting between them.
- No outgoing edges, nor ingoing edges.
- Network is undirective graph.

Graph/Network Structured Data

Database structure diagrams:

- Interconnection of tables.
- A unique property of the table is called key.
- Edge could be in the form of 1:1 1:M, and M:M (note: M = many).
- The relationship between tables (edge) is also determined by the key



Graph Class: Nodes.

```
public class SNPerson  implements Iterable<SNPerson>{
    private  String name;
    private Set<SNPerson> friends;

    public SNPerson(String nm){
        this.name = nm;
        this.friends = new HashSet<SNPerson>();
    }
    public String getName() { return name; }
    public void addFriend(SNPerson fr) {friends.add(fr); }
    public void removeFriend(SNPerson fr) {friends.remove(fr); }
    public boolean hasFriend(SNPerson fr) { return friends.contains(fr); }
    public Iterator<SNPerson> iterator() { return friends.iterator(); }
```

Graph Class: Nodes.

private String name and **private** Set<SNPerson> friends – property to store node and link.

• **public** String getName() { **return** name; }, **public** boolean hasFriend(SNPerson fr) { **return** friends.contains(fr); } - get() method for query purpose.

• **public** void addFriend(SNPerson fr) {friends.add(fr); }, **public** void removeFriend(SNPerson fr) {friends.remove(fr); } - set() method for forming link between nodes.

• this.friends = **new** HashSet<SNPerson>() - used in the object constructor to store the unique elements and it doesn't maintain any specific order of elements.

• **public** Iterator<SNPerson> Iterator() (+ Iterable<SNPerson>) – iteration method in an iterable object that allows you to traverse a collection of elements one-by-one.

Graph Class

Establishing Nodes:

```
SNPerson P1 = new SNPerson("Jane");
SNPerson P2 = new SNPerson("Jay");
```

- Establishing Links:

```
P1.addFriend(P2);
```


```
P2.addFriend(P1);
```

- Program Outcome:
- Comment:

Not scalable, write both lines of the code for bidirectional link relationship only.

Graph Link Improvement

Improving the links between two nodes (1st method):

```
public void addFriend(SNPerson fr) {friends.add(fr); }

SNPerson P1 = new SNPerson(“Jane”);
SNPerson P2 = new SNPerson(“Jay”);
P1.addFriend(P2);
fr.addFriend(this);
```

- Program Outcome:
- Comment: This method calls itself repeatedly. It is a recursive loop.

Graph Link Improvement

Improving the links between two nodes (2nd method):

```
public void addFriend(SNPerson fr) {friends.add(fr); }
SNPerson P1 = new SNPerson("Jane");
SNPerson P2 = new SNPerson("Jay");
P1.addFriend(P2);

public void addFriendship(SNPerson fr){
        this.friends.add(fr);
        fr.friends.add(this);
}
```

- Program Outcome:
- Comment : Still calls for another method than the predetermined method.

Graph Link Improvement

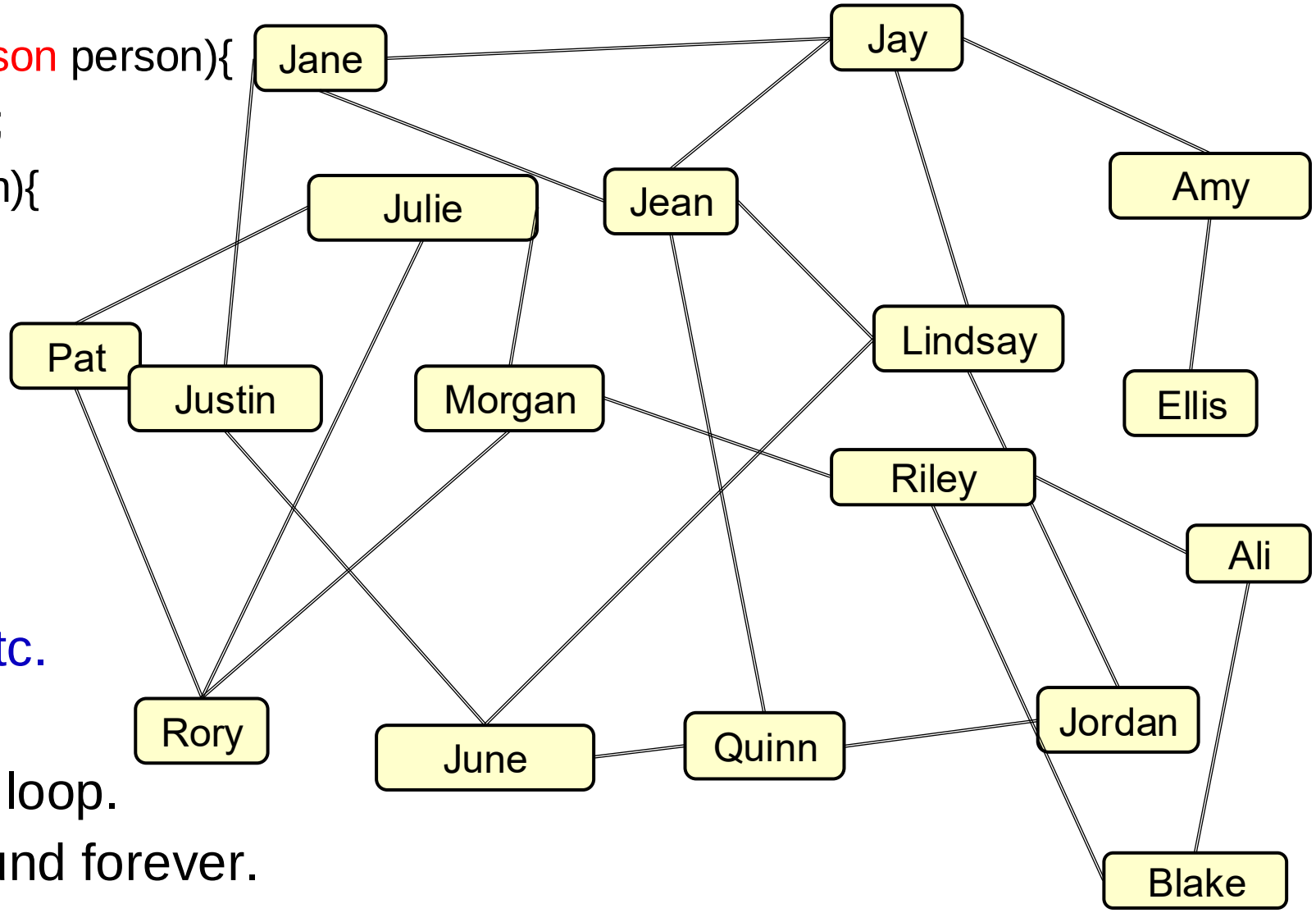
Improving the links between two nodes (3rd method):

```
public void addFriend(SNPerson fr) {friends.add(fr); }
SNPerson P1 = new SNPerson("Jane");
SNPerson P2 = new SNPerson("Jay");
P1.addFriend(P2);

public void addFriendship(SNPerson fr){
        this.friends.add(fr);
        fr. addFriend(this);

}
```

- Program Outcome:
- Comment : Still calls for another method than the predetermined method.

Traversing Graphs

```
/** Print all people in network of a Person  (Buggy) */
public void printNetwork(SNPerson person){
    UI.println(person.getName());
    for (SNPerson friend : person){
        printNetwork(friend);
    }
}
```



Program Outcome:

Jane, Jay, …, Jane, Jay, … etc.

Comment:

- The program is a recursive loop.
- The cycles mean we go round forever.

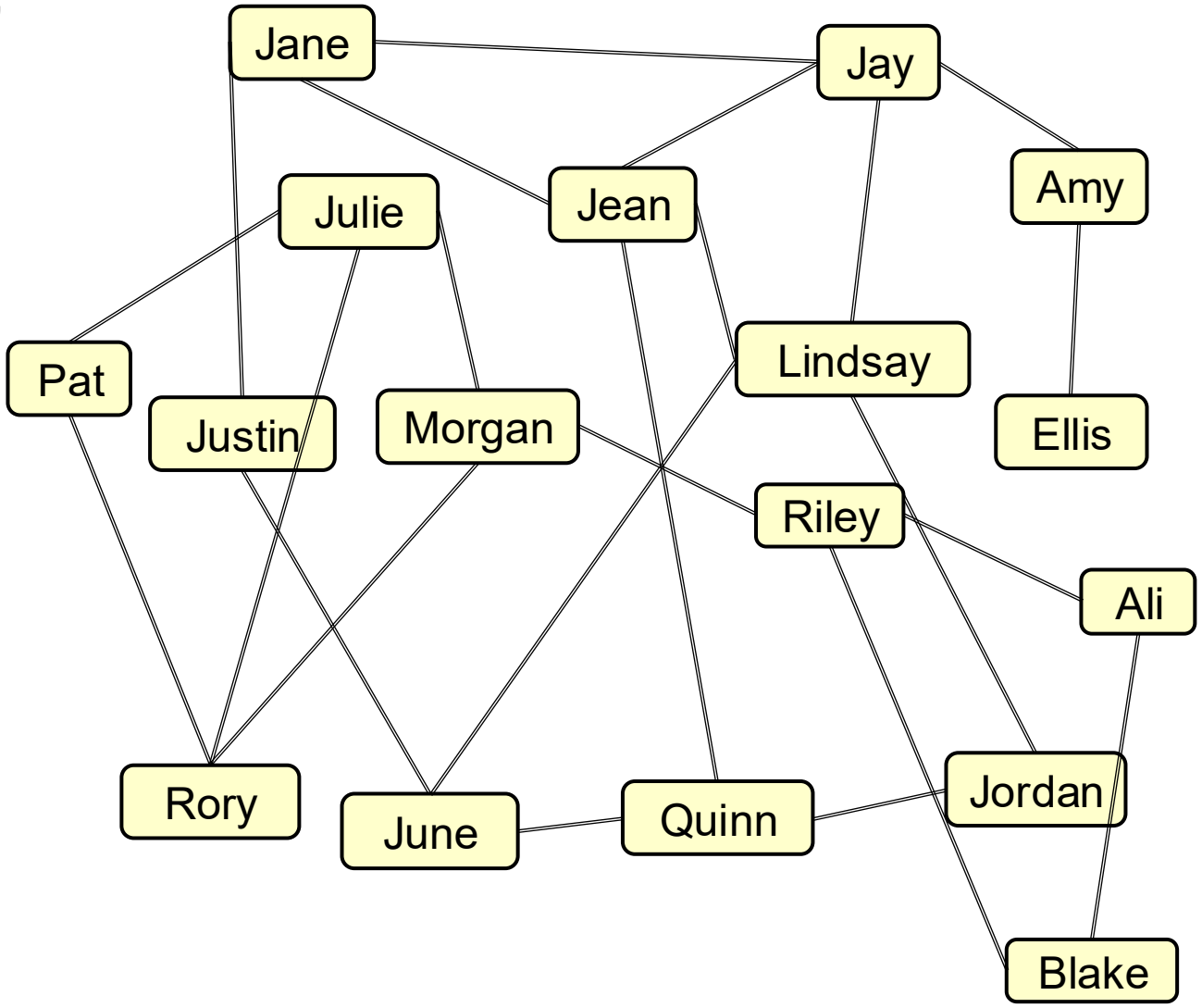
Traversing Graphs: marking nodes

Need to mark the nodes we have visited, and not re-visit them

```
/** Print all people in network of a Person */

public void printNetwork(SNPerson person){
    UI.println(person.getName());
    // mark person as visited
    for (SNPerson friend : person){
        // if the friend is not visited, then
        printNetwork(friend);
    }
}
```

How do we mark nodes?

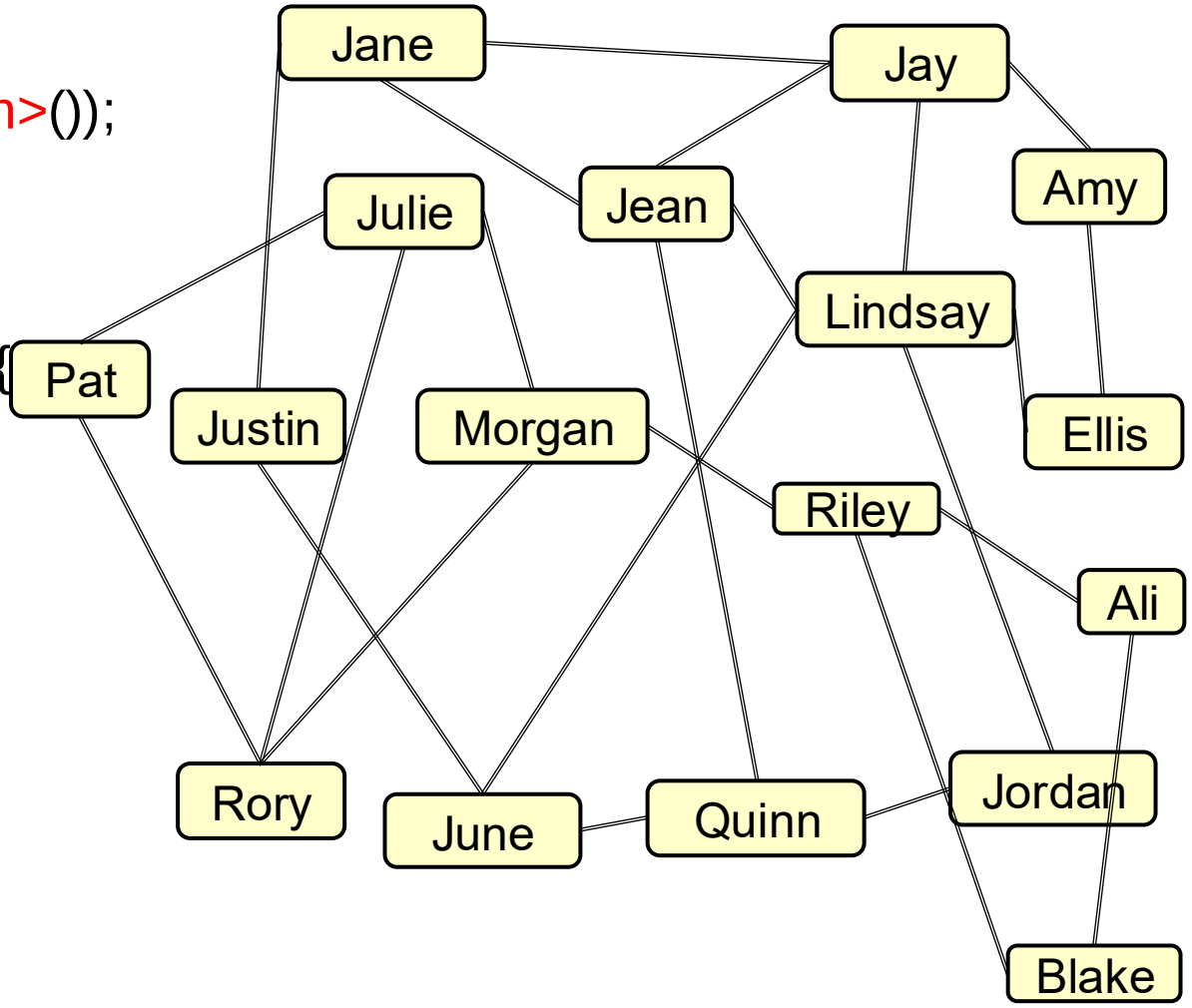
- Keep a Set of nodes we have visited, or
- Store a visited flag in the node



Traversing Graphs: Set of visited nodes

Keep Set of nodes we have visited

```
public void printNetwork(SNPerson person){
    printNetwork(person, new HashSet<SNPerson>());
}

public void printNetwork(SNPerson person,
                         Set<SNPerson> visited){
    UI.println(person.getName());
    visited.add(person);
    for (SNPerson friend : person){
        if (! visited.contains(friend) ){
            printNetwork(friend, visited);
        }
    }
}
```



Still doesn't work if the graph is not connected!!

Traversing Graphs: Set of visited nodes

- Program Outcome:

  Jane, Jay, Amy, Ellis, Jean, Lindsay, Jordan, Quinn, June, and Justin.

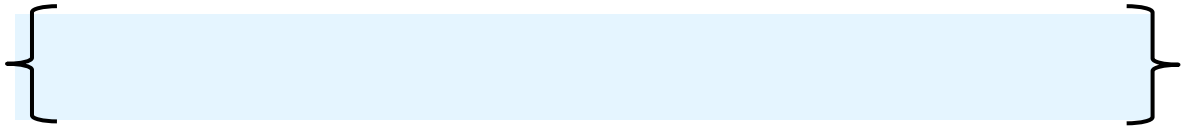
Comment:

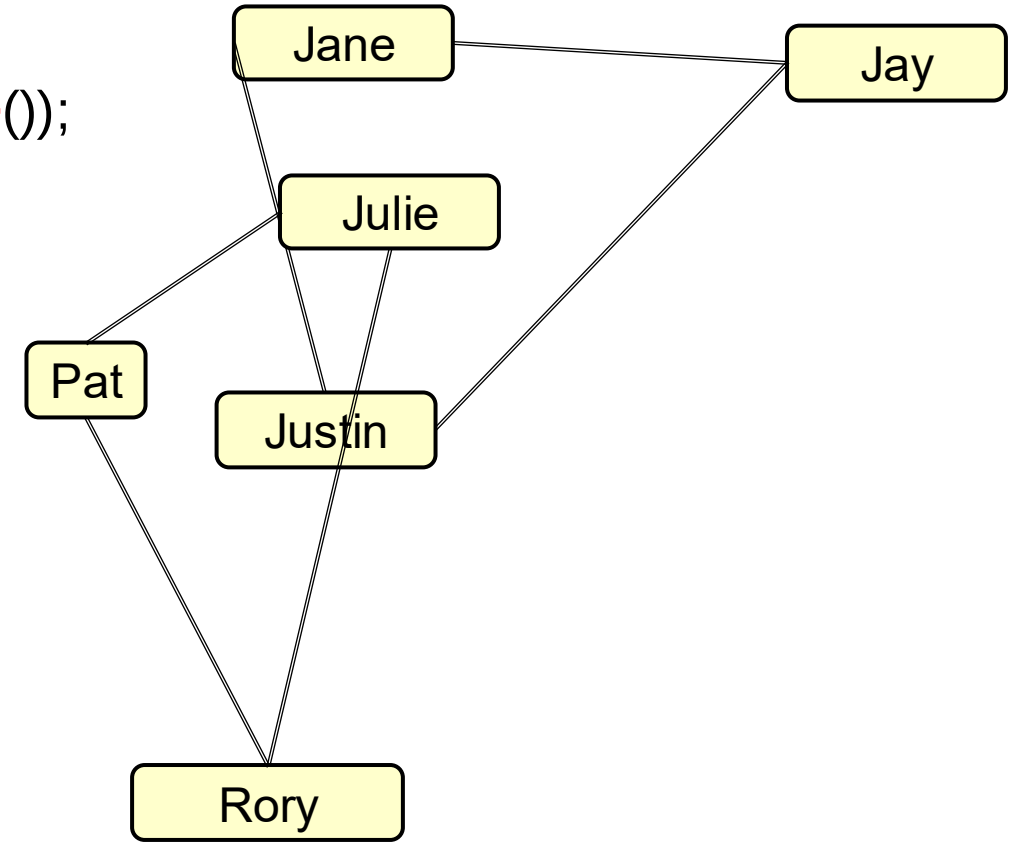
- It shows all connected nodes in a given graph.
- If we start from node Jane, it shows all links in the given graph connected tonode Jane.
- What about nodes Pat, Julie, Rory, Morgan, Riley, Ali and Blake? It still doesn't work, especially since the graph is not connected!

Set of visited nodes

Keep Set of nodes we have visited

```
public void printNetwork(SNPerson person){
    printNetwork(person, new HashSet<SNPerson>());
}

public void printNetwork(SNPerson person,
                         Set<SNPerson> visited){
    UI.println(person.getName());
    visited.add(person);
    for (SNPerson friend : person){
        if (! visited.contains(friend) ){
            printNetwork(friend, visited);
        }
    }
}
```





Still doesn't work if the graph is not connected!!

Set of visited nodes

Program outcome:

Jane, Jay, and Justin.

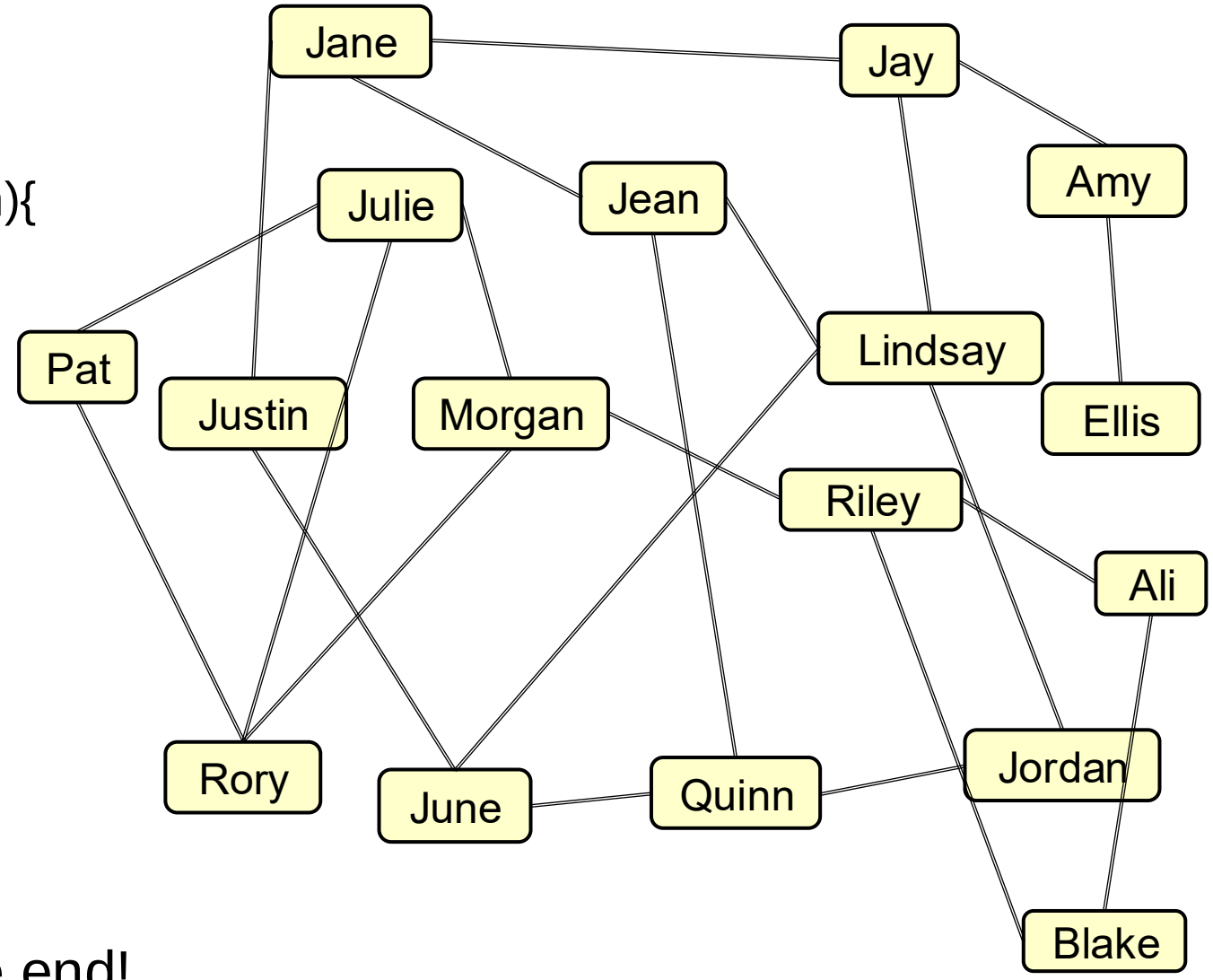
Comment:

- It shows all connected nodes in a given graph.
- It still does not cover the nodes connected in the other graph – need to establish the total number of graphs and how many connected nodes in each graph before performing the algorithm.
- It doesn’t work if the graph is not connected! (e.g. Pat, Julie and Rory nodes are not included).
- Need to calculate the total number of nodes.

Traversing Graphs: visited flag inside node

Visited flag inside the node:

```
/** Print all people in network of a Person */

public void printNetwork(SNPerson person){
    UI.println(person.getName());
    person.visit();
     for (SNPerson friend : person){
        if ( ! friend.isVisited()){
            printNetwork(friend);
        }
    }
}
```



Need to reset all the visited flags at the end!

Traversing Graphs: Visited Flag Inside Node

Program outcome:

Jane, Jay, Amy, Ellis, Jean, Lindsay, Jordan, Quinn, June, and Justin.

Comment:

- It shows all connected nodes in a given graph.
- Need to calculate the total number of nodes.
- If you need to start traversing the graph all over again, we need to reset all the visited flags at the end!

Graph Nodes (with visited flag)

```
public class SNPerson  implements Iterable<SNPerson>{
    private  String name;
    private Set<SNPerson> friends;
    private boolean visited; //This is the flag

    public SNPerson(String nm){
        this.name = nm;
        this.firends = new HashSet<SNPerson>();
    }
    public String getName() { return name; }
    public void addFriend(SNPerson fr)       { friends.add(fr); }
    public void removeFriend(SNPerson fr) { friends.remove(fr); }
    public boolean hasFriend(SNPerson fr) { return friends.contains(fr); }
    public Iterator<SNPerson> iterator()      { return friends.iterator(); }

    public void visit()     {visited=true; }
    public void unvisit() {visited=false; }
    public boolean isVisited() {return visited; }
```

Traversing Graph (with Visited Flag)

For traversing graph with visited flag method, modification of the original program:

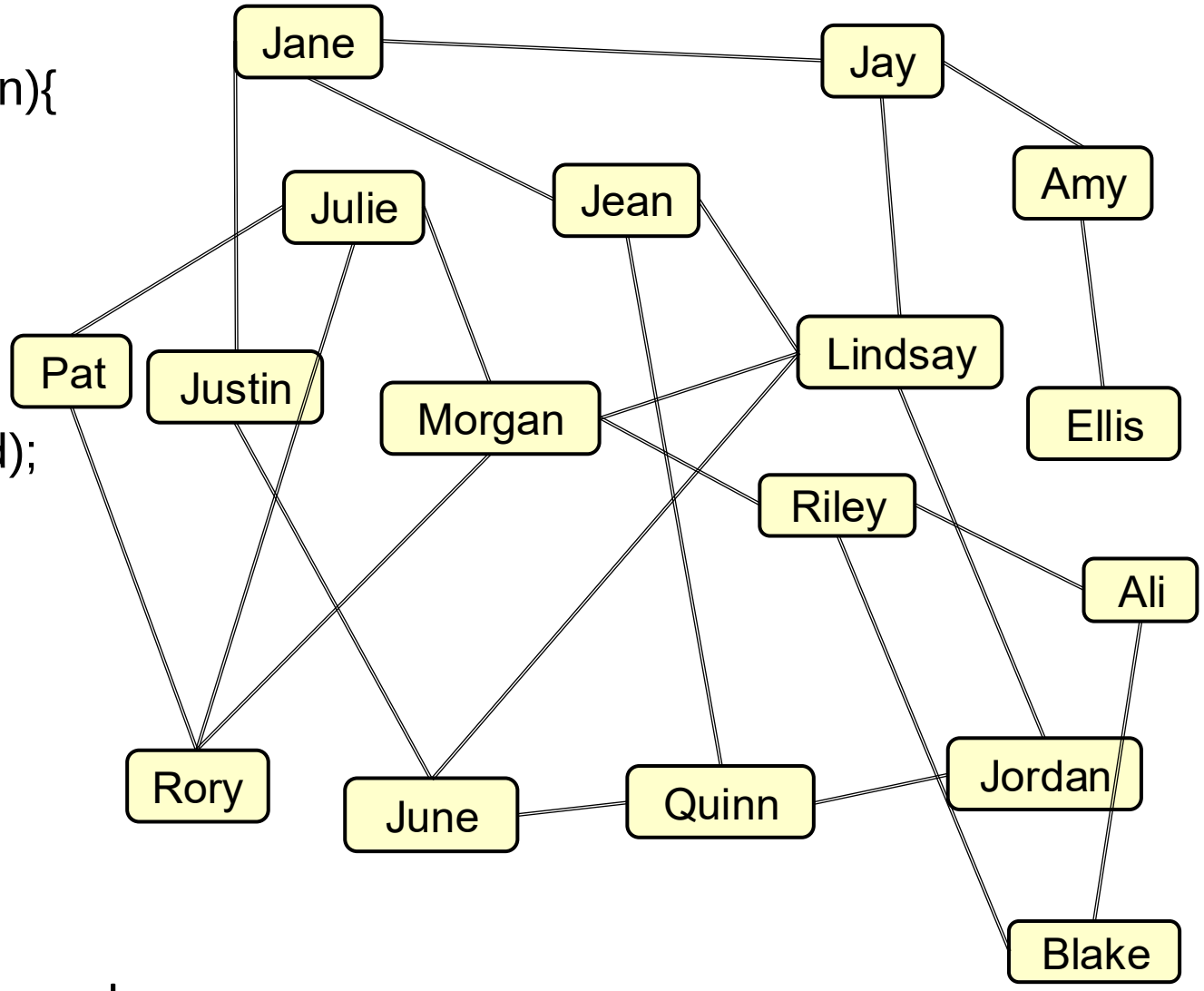
- **private** boolean visited – property to store the visited flag.

- **public** boolean isVisited() {**return** visited; } – get() method for status of visited flag.

- **public** void visit() {visited=true; } and **public** void unvisit() {visited=false; } – set() method to set or remove the visited flag, no need for the helper methods that we have had before.

Traversing Graphs : Count Connected Nodes

```
/** Find number of friends in network */

public int countConnected(SNPerson person){
    person.visit();
    int count =1;
    for (SNPerson friend : person){
        if ( ! friend.isVisited()) {
            count += countConnected(friend);
        }
    }
    return count;
}
```



Traversing a graph makes a tree within the graph.

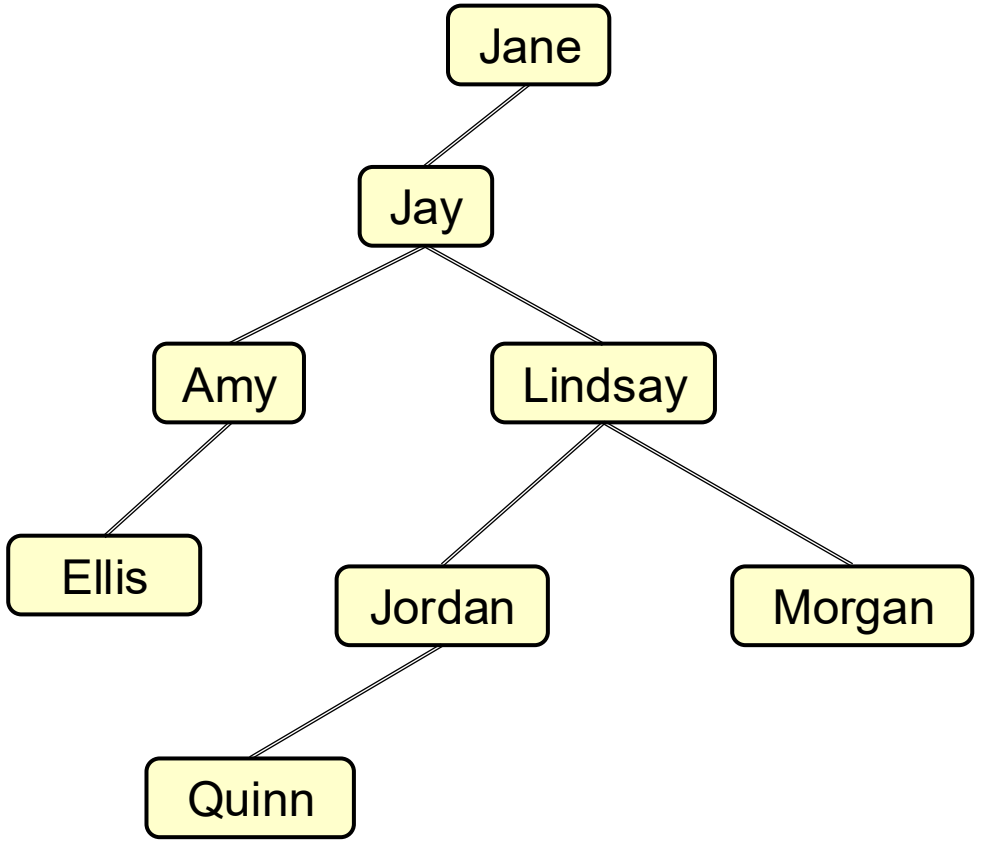
Traversing Graphs : Count Connected Nodes

Program Outcome:

- No of friends: XX

Comment:

- For counting connected nodes, it is like
- Calculating the number of paths in the graph
- The traversing process is like searching the trees.
- Traversing a graph makes a tree within the graph.
- Checking the length (depth first) and width (breadth first) of the path – finding the shortest path.



Traversing Graphs: Connected To (Recursive)

```
/** Are two people connected in the network */

public boolean connectedTo(SNPerson person, SNPerson query){
    if (person.equals(query) ) {
        return true;
    }
    person.visit();
    for (SNPerson friend : person){
        if ( ! friend.isVisited()  &&  connectedTo(friend, query) ) {
            return true;
        }
    }
    return false;
}
```

Note: need to reset all the visited flags before you call this!

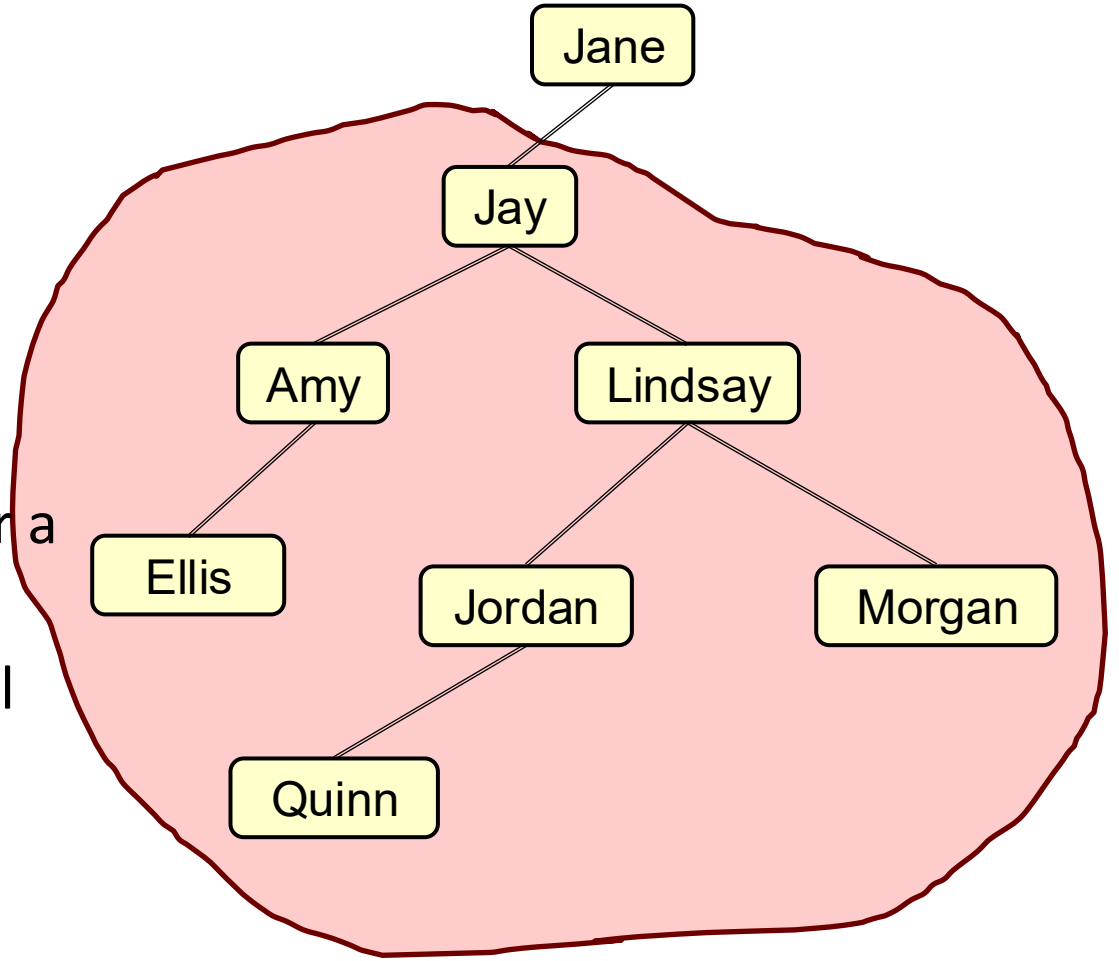
Traversing Graphs : Connected To (Recursive)

Program Outcome:

Connected or Not Connected: XX

Comment:

- We set the visited flag.
- Check all subgraph of the node that comes after a given node that we start with.
- Need to reset all the visited flags before you call this method.



Traversing Graphs: Connected To

```
/** Are two people connected in the network */

public boolean connectedTo(SNPerson person, SNPerson query){
    return connectedTo(person, query, new HashSet<SNPerson>());
}

public boolean connectedTo(SNPerson person, SNPerson query, Set<SNPerson> visited){
    if (person.equals(query) ) {
        return true;
    }
    visited.add(person);
    for (SNPerson friend : person){
        if ( ! visited.contains(friend)  &&  connectedTo(friend, query, visited) ) {
            return true;
        }
    }
    return false;
}
```

Traversing Graphs : Connected To (Recursive)

Program Outcome:

Connected or Not Connected: XX

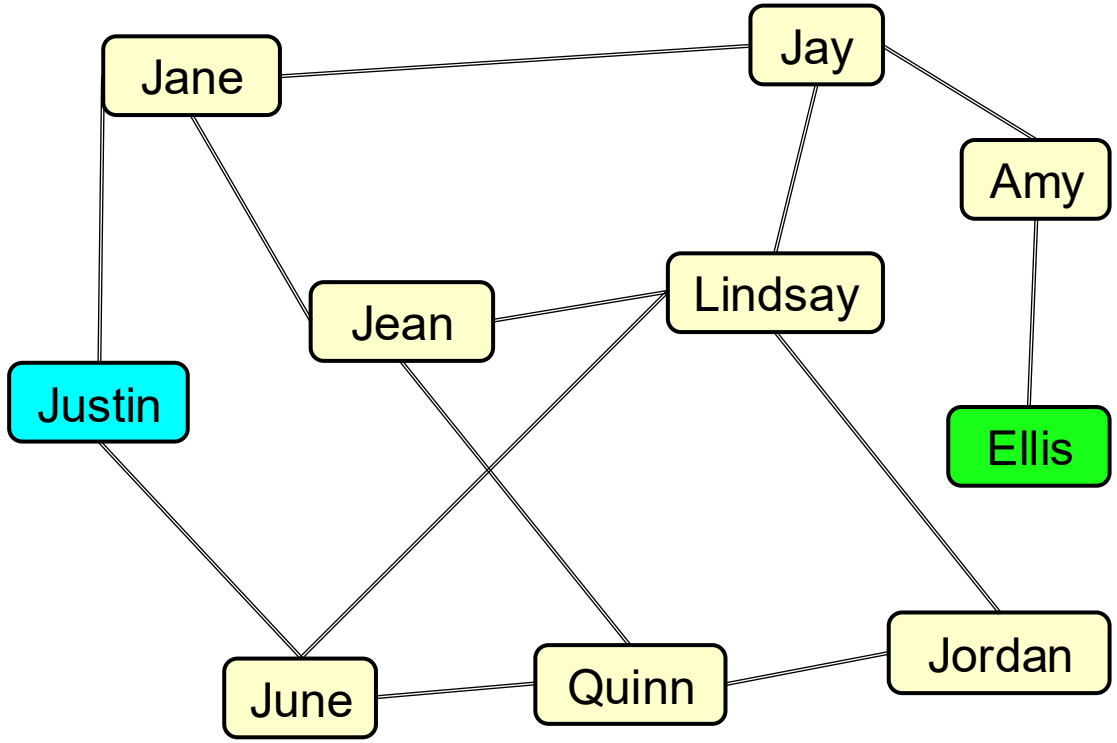
Comment:

- We have a set of visited person (i.e. new HasSet<SNPerson>()).
- Helper method
  - connectedTo(SNPerson person, SNPerson query, Set<SNPerson> visited)

Traversing Graphs: connectedTo

```
/** Are two people connected in the network */
public boolean connectedTo(SNPerson person, SNPerson query){
    return connectedTo(person, query, new HashSet<SNPerson>());
}
public boolean connectedTo(SNPerson person, SNPerson query, Set<SNPerson> visited){
    UI.println(person);
    if (person.equals(query) ) {  return true; }
    visited.add(person);
    boolean ans = false;
    for (SNPerson friend : person){
        if ( ! visited.contains(friend)  &&
                connectedTo(friend, query, visited) ) {
            ans = true;
        }
    }
    visited.remove(person);
    return ans;
}
```

What happens if we unvisited the node here?



Traversing Graphs : Connected To (Recursive)

Program Outcome:

Connected or Not Connected: XX

Comment:

- Check if Justin is connected to Ellis.
- `visited.remove(person)` – It allows the algorithm to traverse all possible paths to connect Justin with Ellis.
- It is used for solving the bus and maze problems in the Assignment 6 – find all possible paths to solve the bus routes and maze.

Traversing Graphs: Connected To (iterative)

```
/** Are two people connected in the network */

public boolean connectedTo(SNPerson person, SNPerson query){
    Stack<SNPerson> stack = new ArrayDeque<SNPerson>();
    Set<SNPerson> visited = new HashSet<SNPerson>();
    stack.push(person);
    while (! stack.isEmpty()){
        SNPerson p = stack.pop();
        visited.add(p);
        if (p.equals(query) ) { return true; }
        for (SNPerson friend : p){
            if ( ! visited.contains(friend)) { stack.push(friend); }
        }
    }
    return false;
}
```

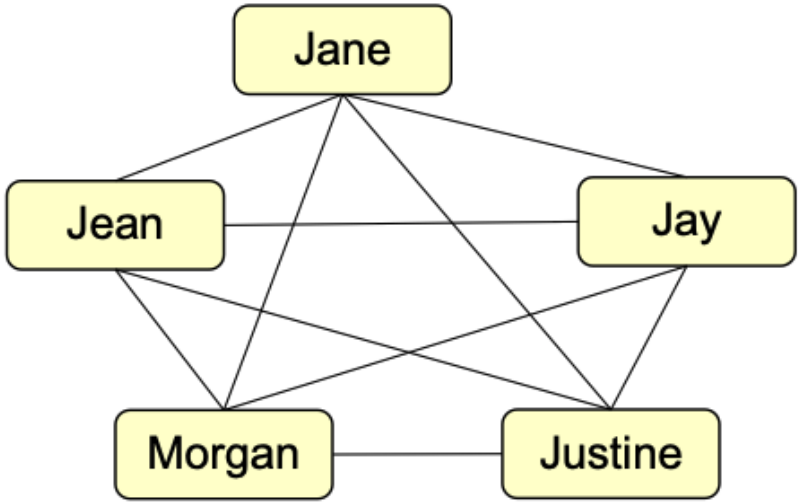
Traversing Graphs: Connected To (iterative)

- Program Outcome:

  Connected or Not Connected: XX

Comment:

- Non-recursive way, but iterative way to find links between nodes.
- Find number of nodes: depth or breadth first.
- Push the node to the stack.
- Check if connected, pop the node from the stack to visited set.
- This algorithm was also used for traversing the trees.

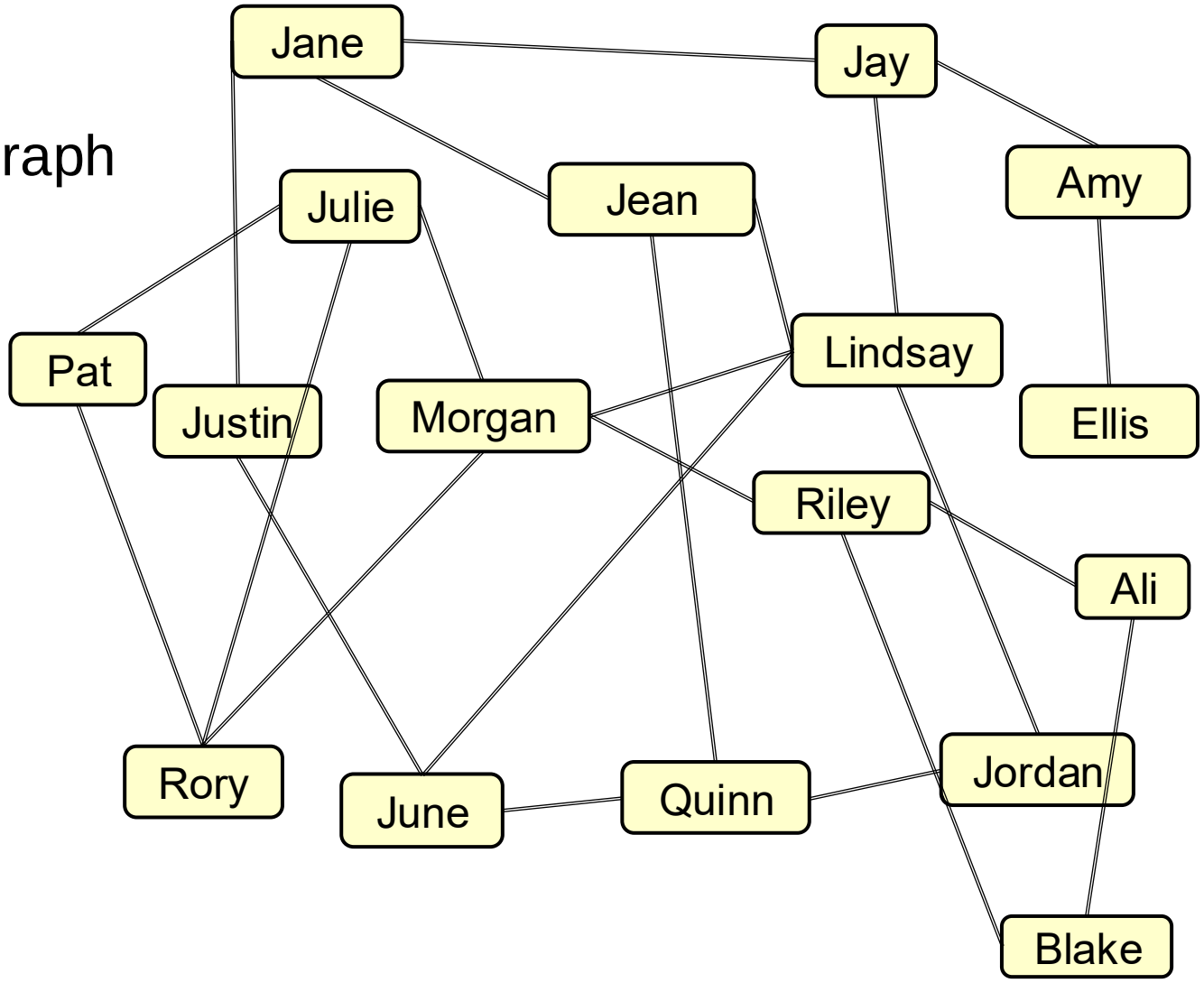


* Process N nodes.
• Fully connected NxN O(N^2)
• Check all neighbours i.e. hops and paths O(N!).

Is Graph Connected?

- How do I represent this graph?
- Just having one node isn't enough!
- Need to store the set of nodes in the graph

```
private Set<SNPerson> graph;
```



Is it connected?

pick a node,

find all the connected nodes

does this cover all the nodes?

Is Graph Connected?

Program Outcome:

Comment:

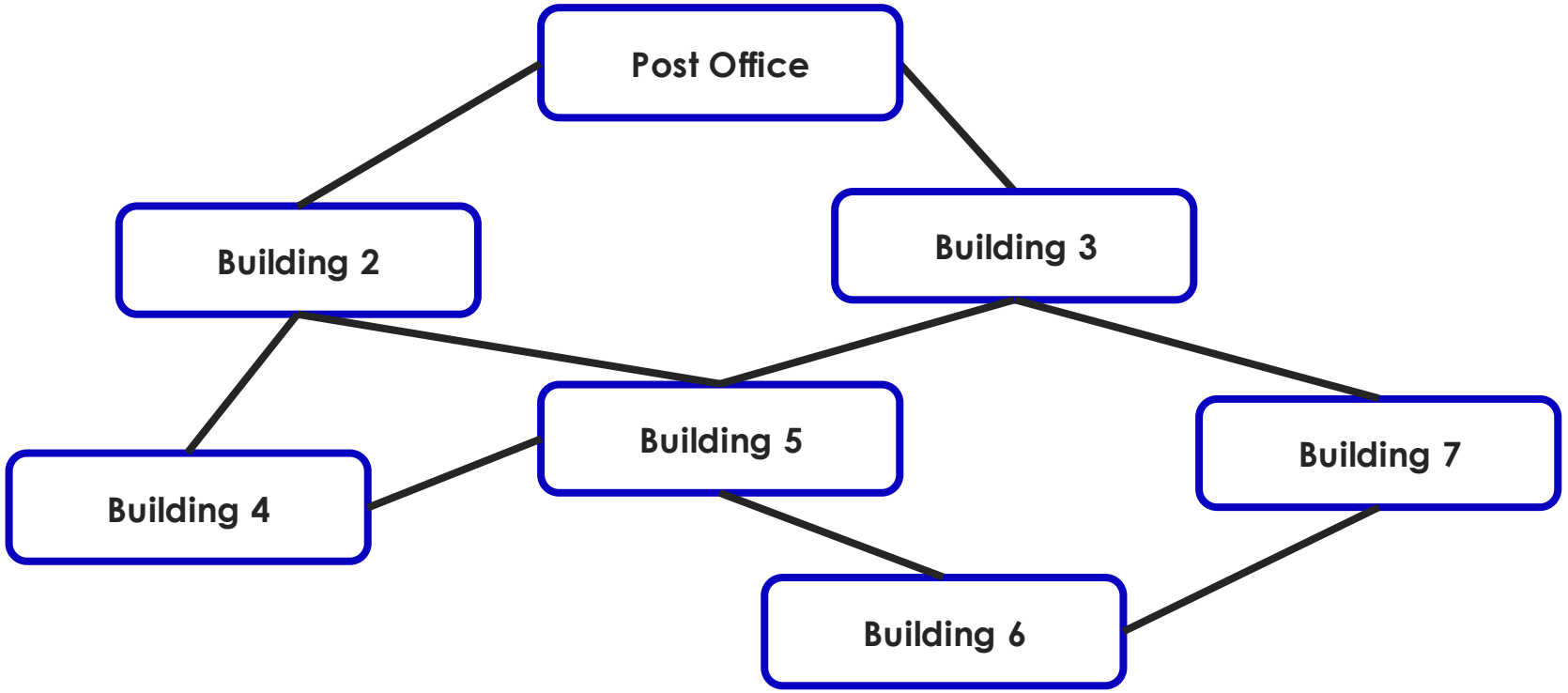
- Count all nodes that exist in the graph – this gives the entire population of the nodes in the graph.
- Start with a node and count all connected nodes in a given graph into a set.
- Check for other connected nodes in a given graph, also into a set (if it exists).

Terms

Term	Meaning
Vertex (Node)	A point/object
Edge	Connection between vertices
Adjacent	Connected directly
Path	Sequence of connected vertices
Cycle	Path returning to start
Directed Graph	One-way edges
Undirected Graph	Two-way edges

Review

Create the post office's route using graph



The output of the program will be:

Neighbours of Building 3: [Post Office, Building 5, Building 7]
Neighbours of Post Office: [Building 2, Building 3]