CMPU 101 § 52 · Computer Science I

Trees

15 February 2023

Where are we?

Now we can see how lists are defined:

```
data List:
  | empty
  | link(first, rest :: List)
end
```

Self-reference

Given this data definition, we can write functions that recursively process a list, following this pattern:

```
fun list-fun(lst :: List) -> ...:
  doc: "Template for a fn that takes a List"
  cases (List) lst:
    | empty => ...
    | link(f, r) =>
      ... f ...
      ... list-fun(r) ...
  end
where:
   list-fun(...) is ...
end
```

Every data definition has a corresponding template.

The more complex the data definition is – lots of variants, recursion, etc. – the more helpful it is to use the template!

Given a (recursive) data definition, you write a template by:

1. Creating a function header
2. Using `cases` to break the data input into its variants
3. In each case, listing each of the fields in the answer
4. Calling the function itself on any recursive fields

Rumor mills

The news [of Emma and Mr. Knightley's engagement] was universally a surprize wherever it spread; and Mr. Weston had his five minutes share of it…

"It is to be a secret, I conclude," said he. "*These matters are always a secret, till it is found out that every body knows them.* Only let me be told when I may speak out.—I wonder whether Jane has any suspicion."

Jane Austen, *Emma*, 1815

He went to Highbury the next morning, and satisfied himself on that point. He told her the news… and Miss Bates being present, it passed, of course, to Mrs. Cole, Mrs. Perry, and Mrs. Elton, immediately afterwards. It was no more than the principals were prepared for; *they had calculated from the time of its being known at Randalls, how soon it be over Highbury*; and were thinking of themselves, as the evening wonder in many a family circle…

Jane Austen, *Emma*, 1815

Tracking rumors

Suppose we want to track gossip in this rumor mill.

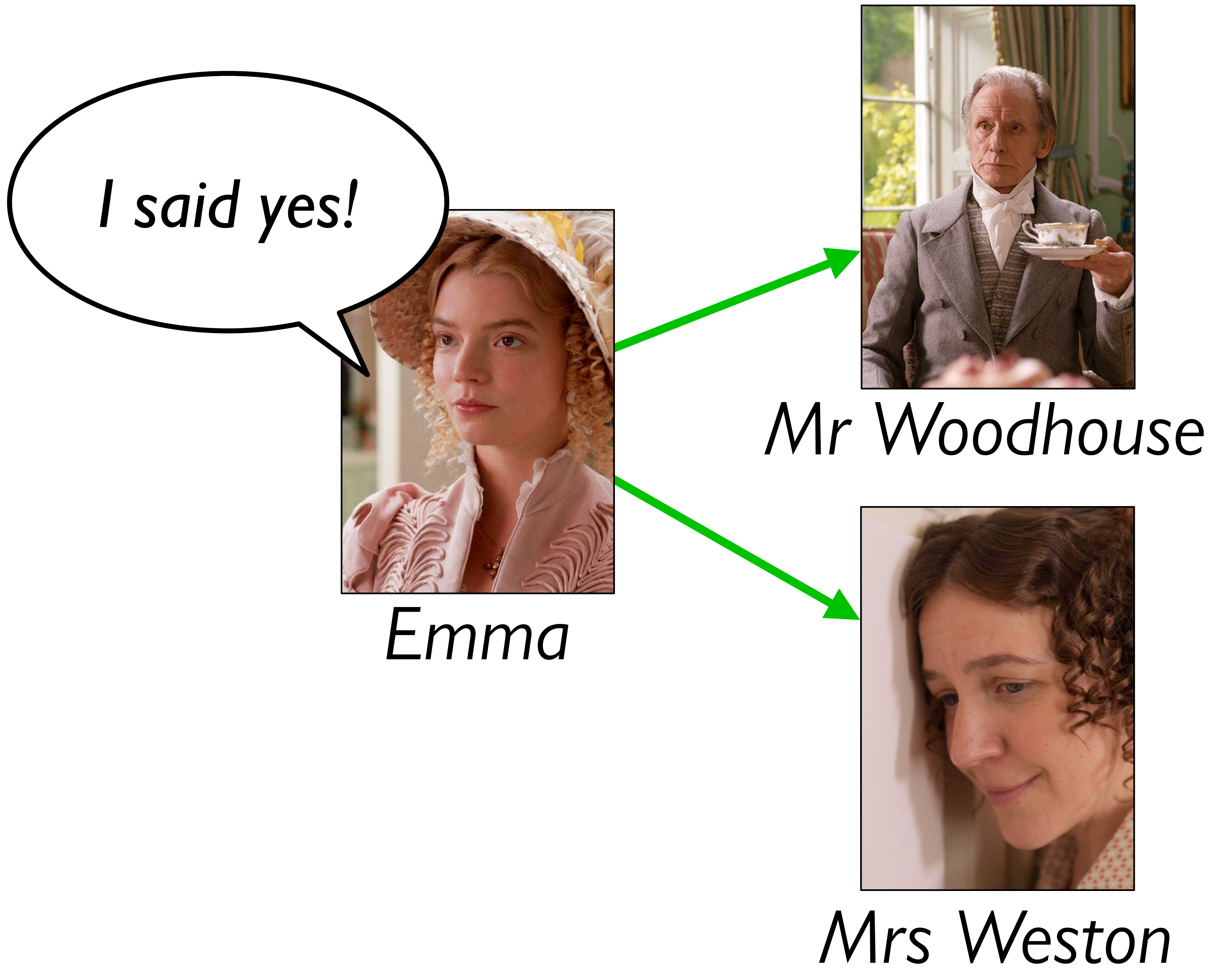
Tracking rumors

Suppose we want to track gossip in this rumor mill.

Emma

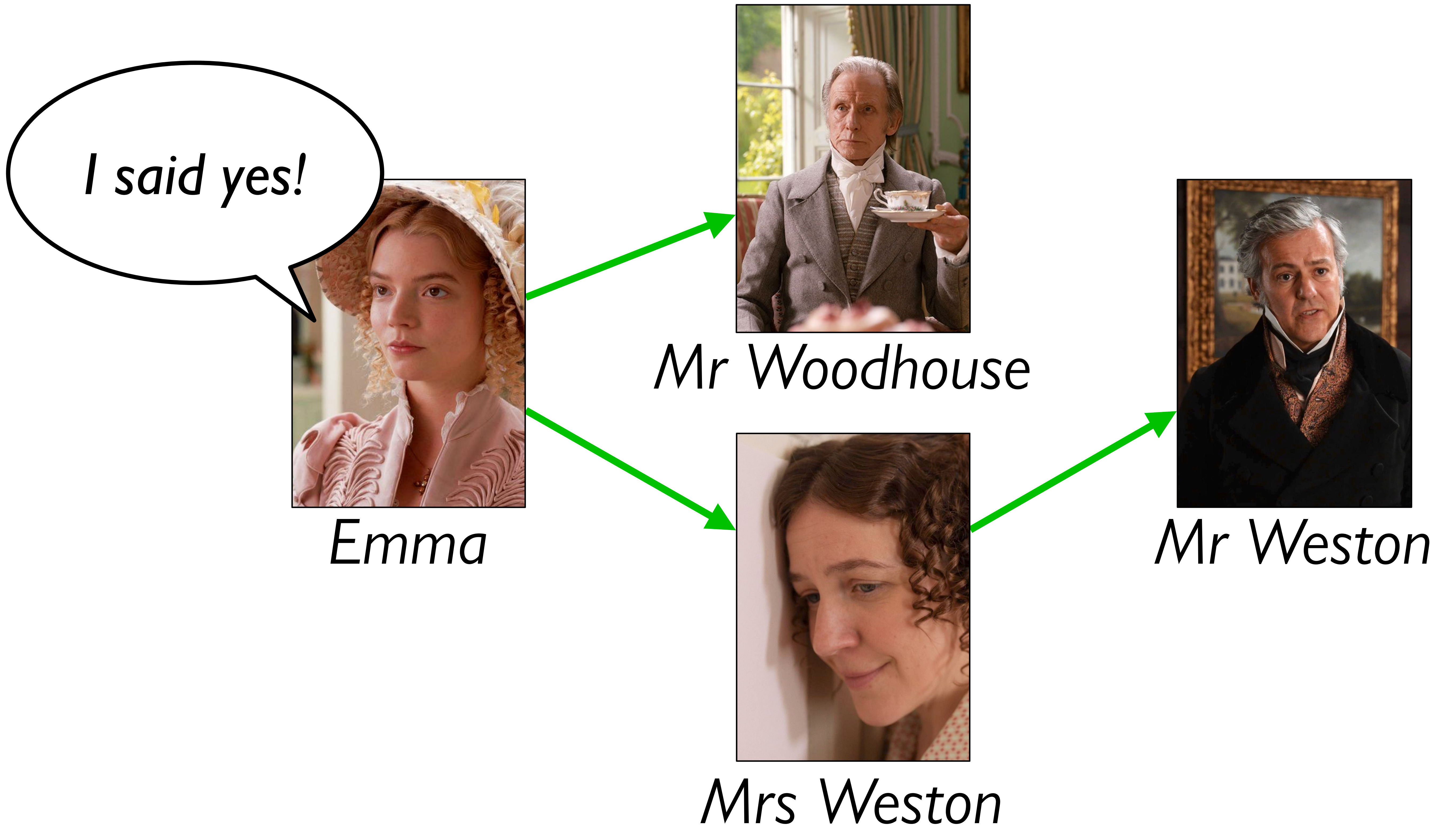
Tracking rumors

Suppose we want to track gossip in this rumor mill.



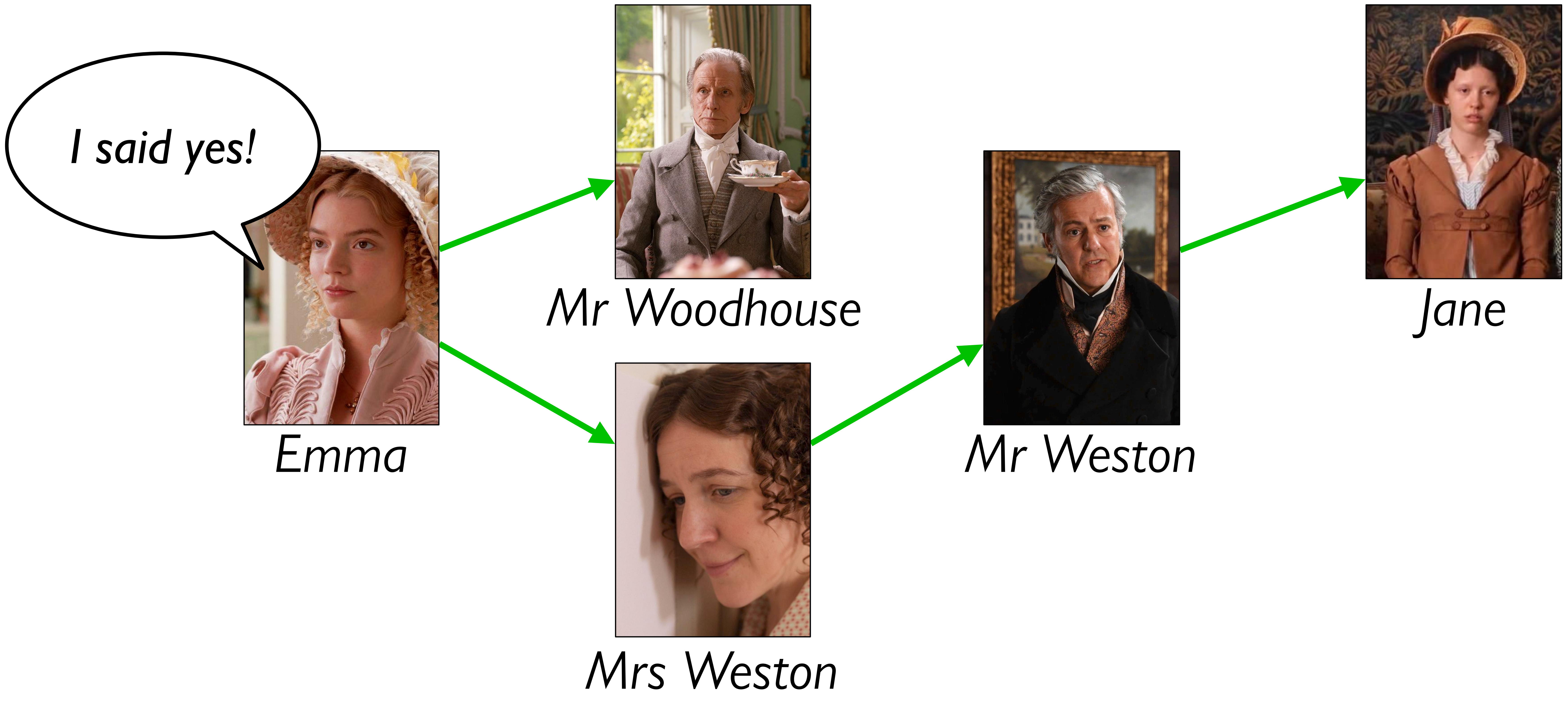
Tracking rumors

Suppose we want to track gossip in this rumor mill.



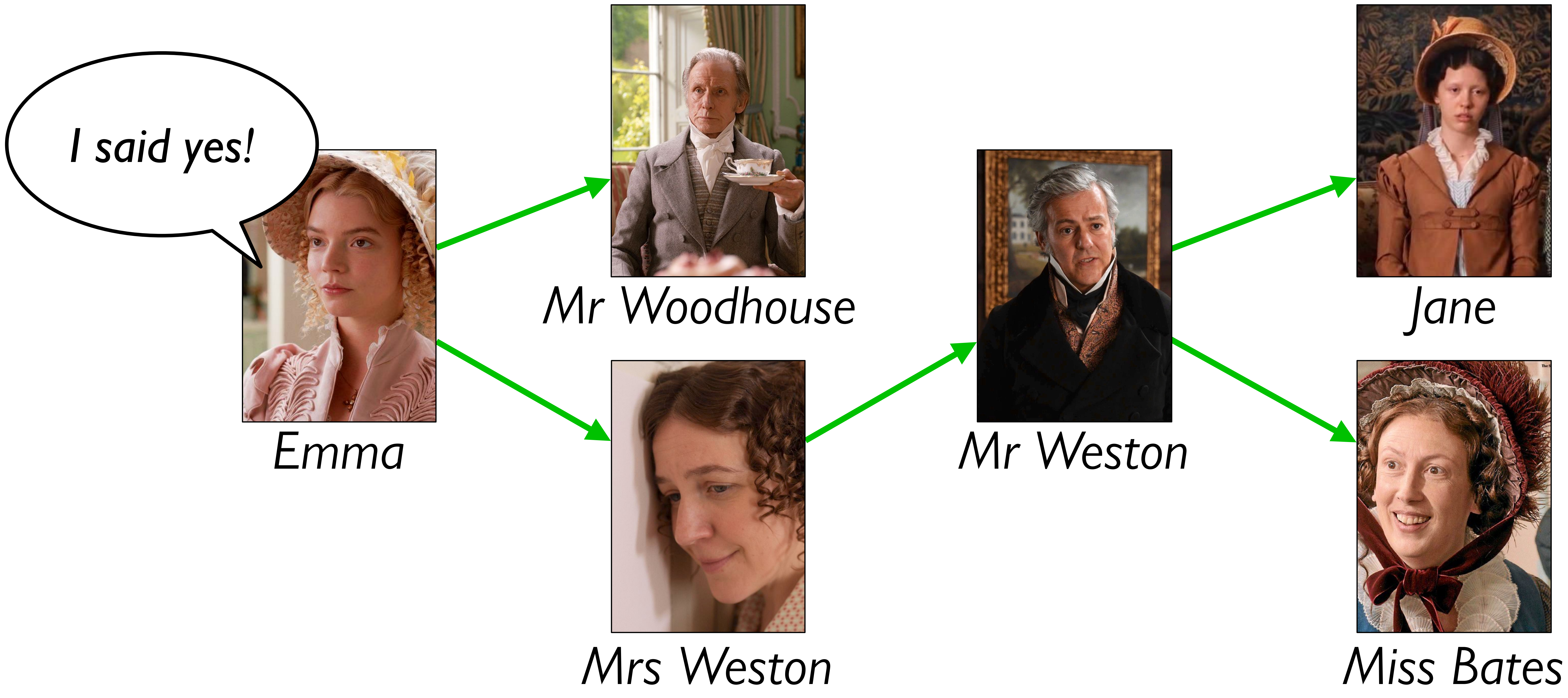
Tracking rumors

Suppose we want to track gossip in this rumor mill.



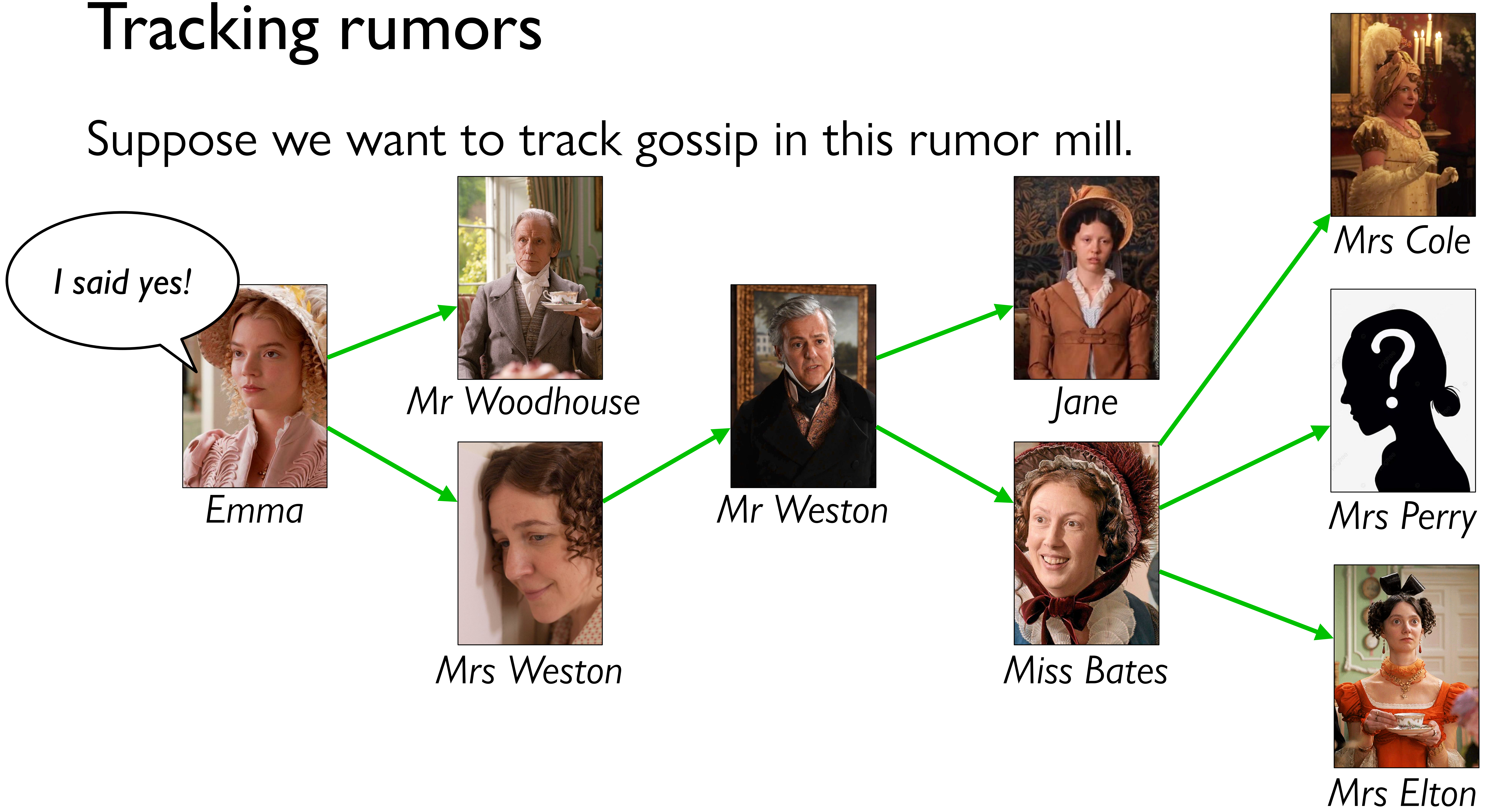
Tracking rumors

Suppose we want to track gossip in this rumor mill.



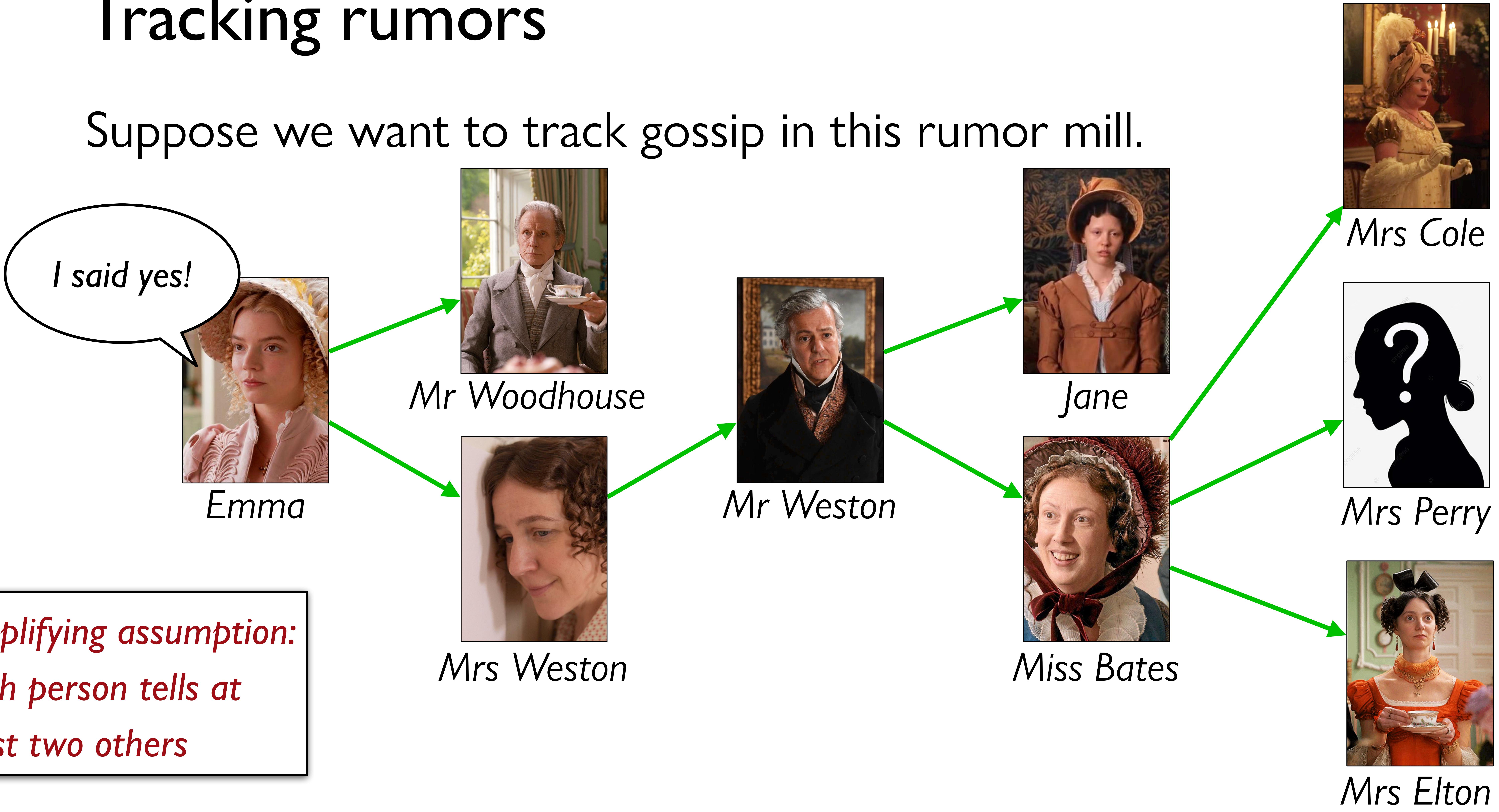
Tracking rumors

Suppose we want to track gossip in this rumor mill.



Tracking rumors

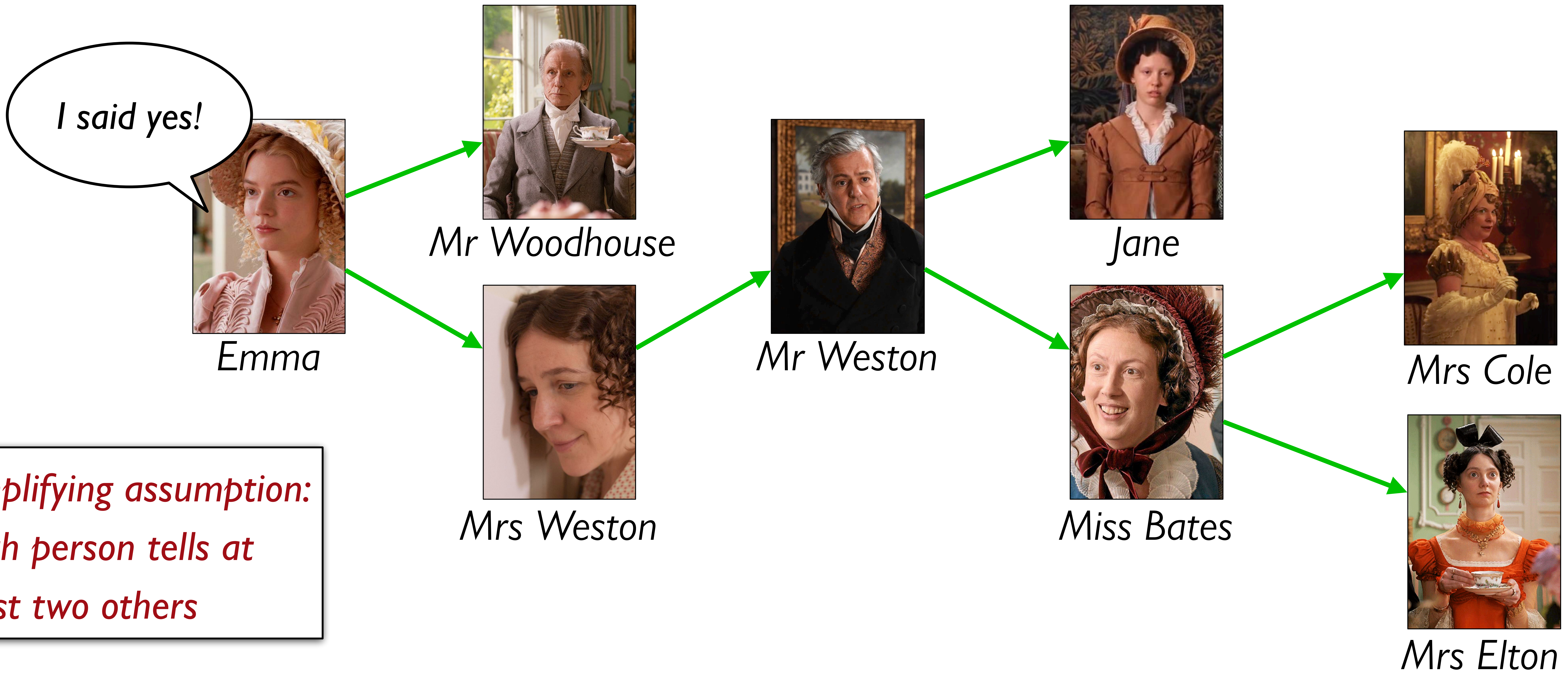
Suppose we want to track gossip in this rumor mill.



Simplifying assumption: Each person tells at most two others

Tracking rumors

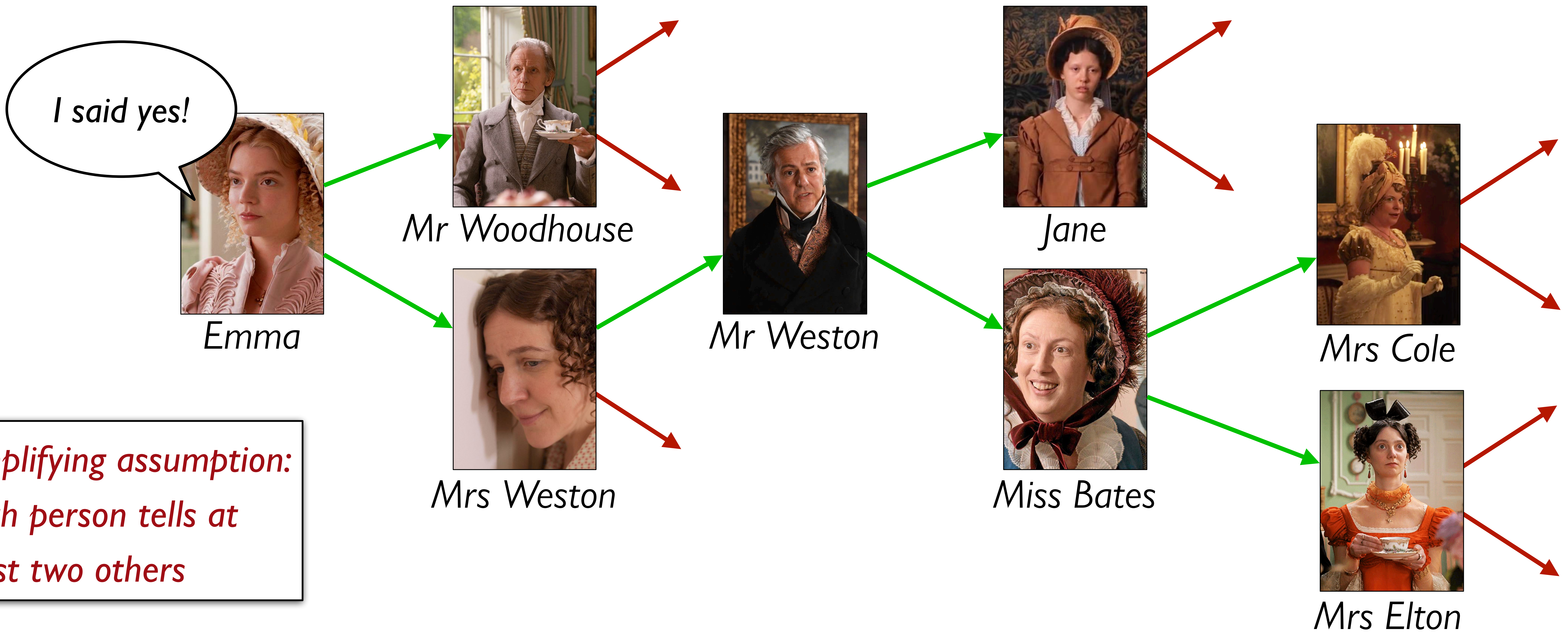
Suppose we want to track gossip in this rumor mill.



Simplifying assumption: Each person tells at most two others

Tracking rumors

Suppose we want to track gossip in this rumor mill.

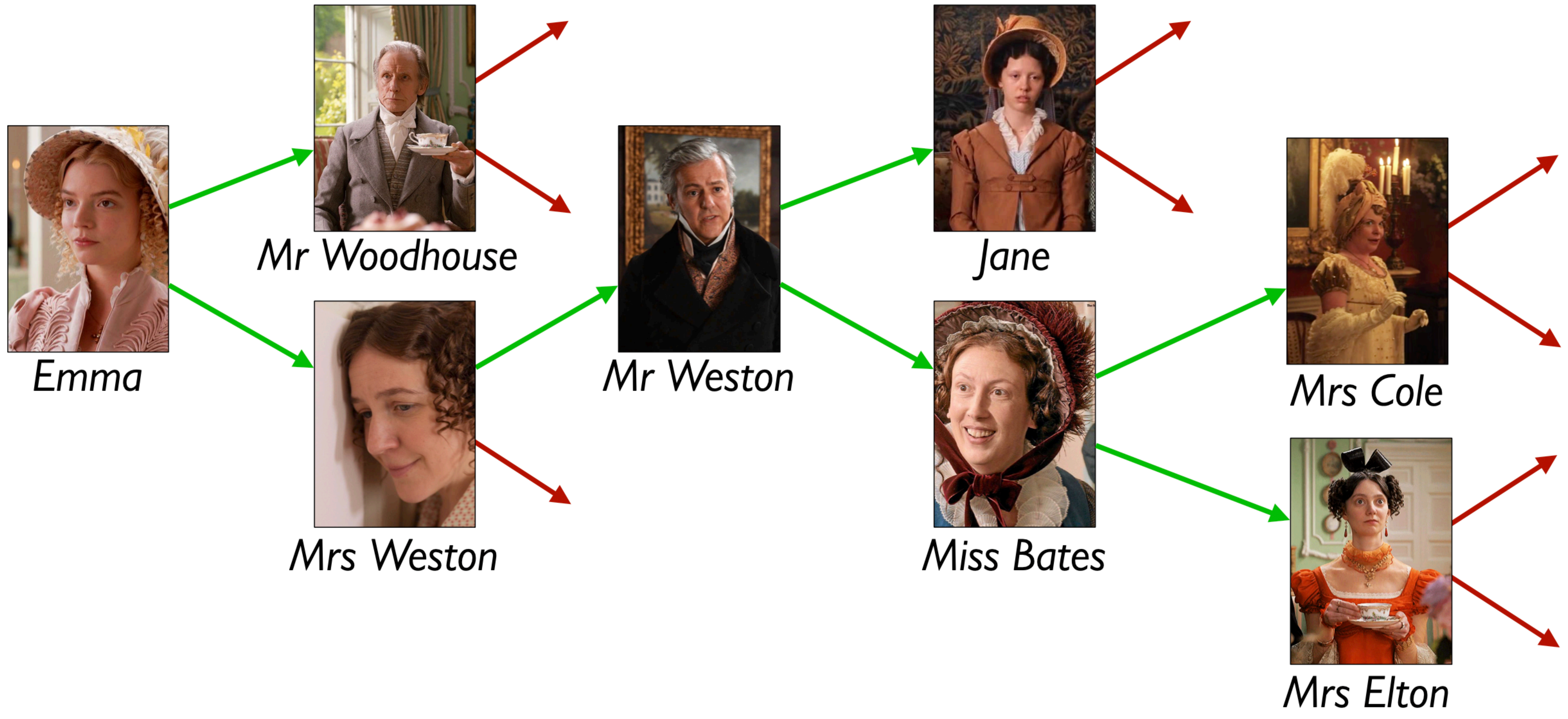


Simplifying assumption: Each person tells at most two others

If you ignore the silly Jane Austen example, this is a pretty serious problem.

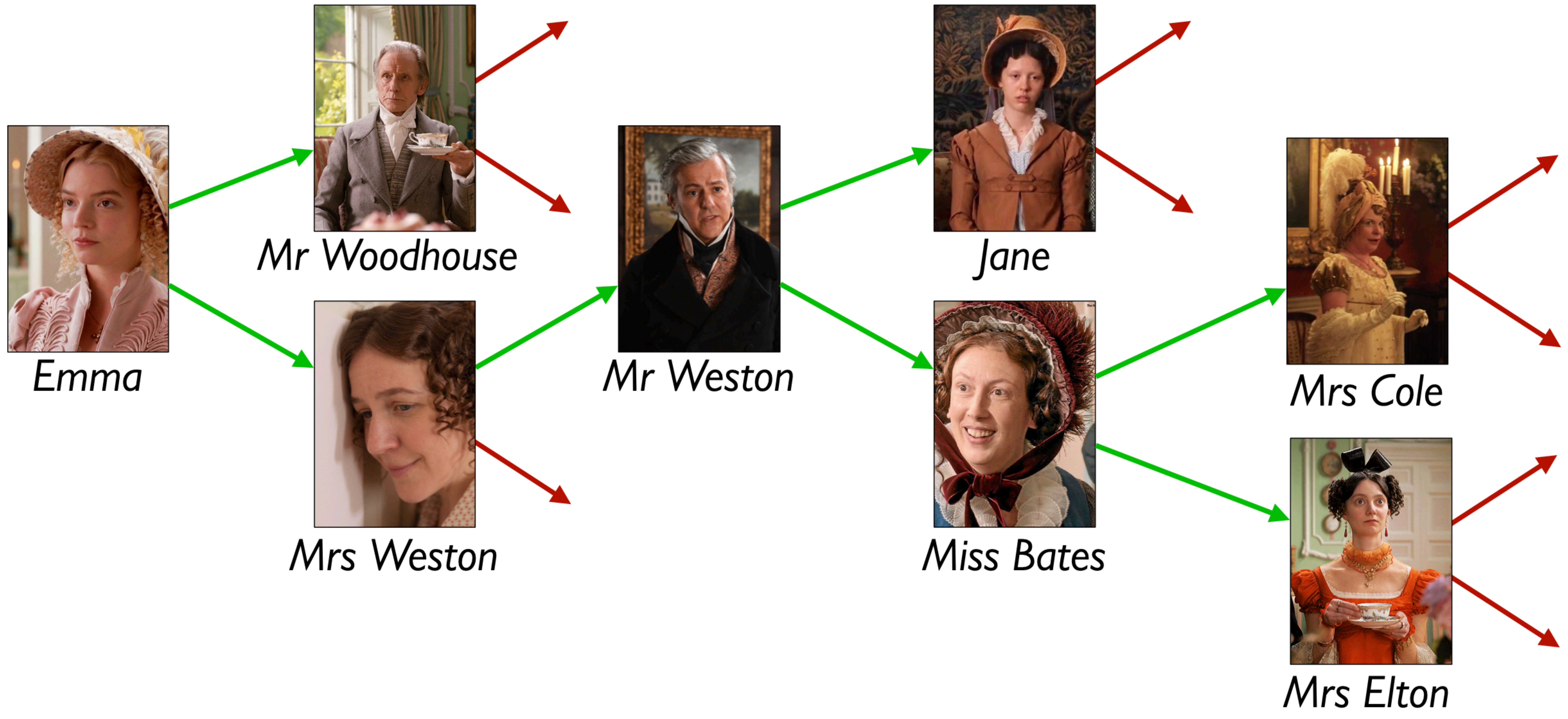
A lot of research right now is focused on building models of how information – and misinformation! – spreads through social networks, both in person and online.

Representing rumor mills



Is a rumor mill simply a list of people?

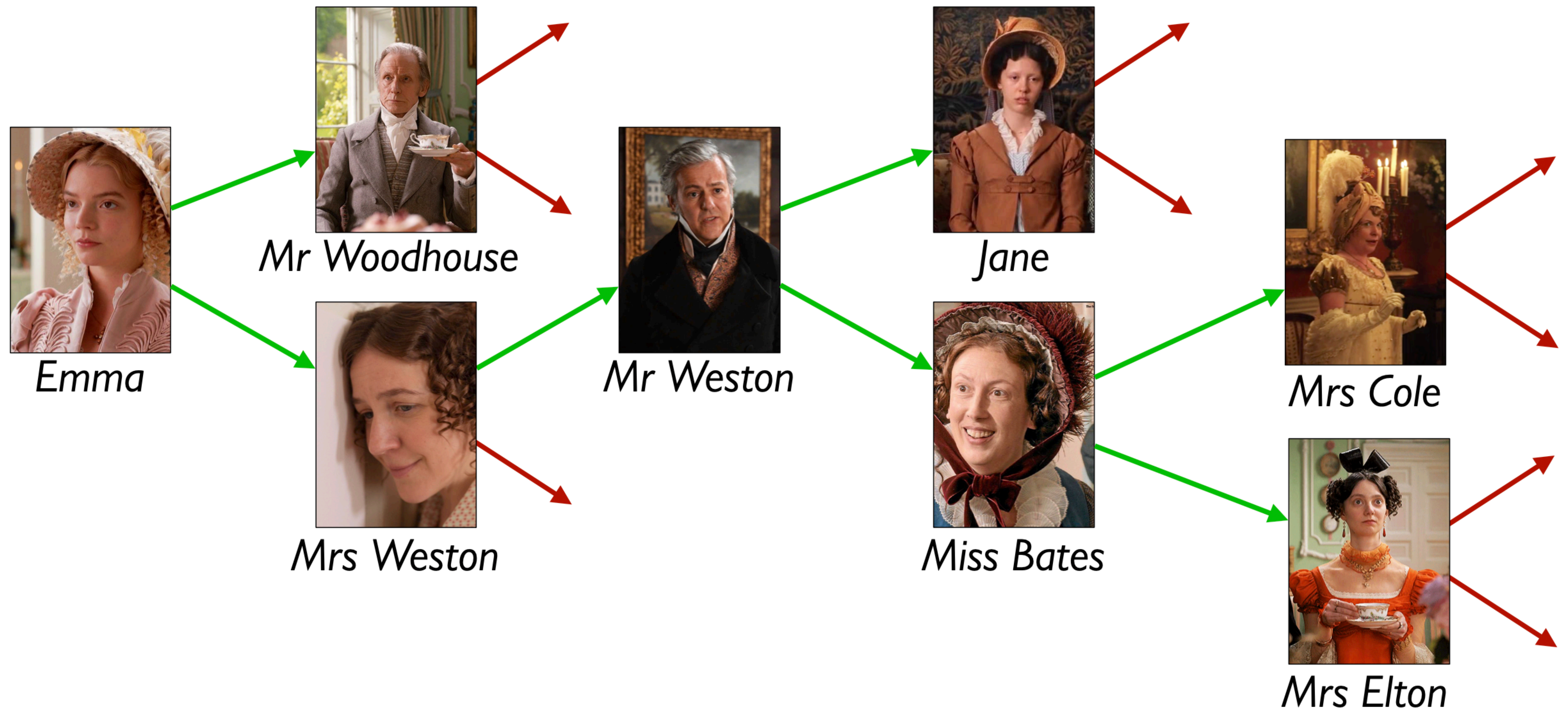
Representing rumor mills



Is a rumor mill simply a list of people?

No, because there are relationships among the people.

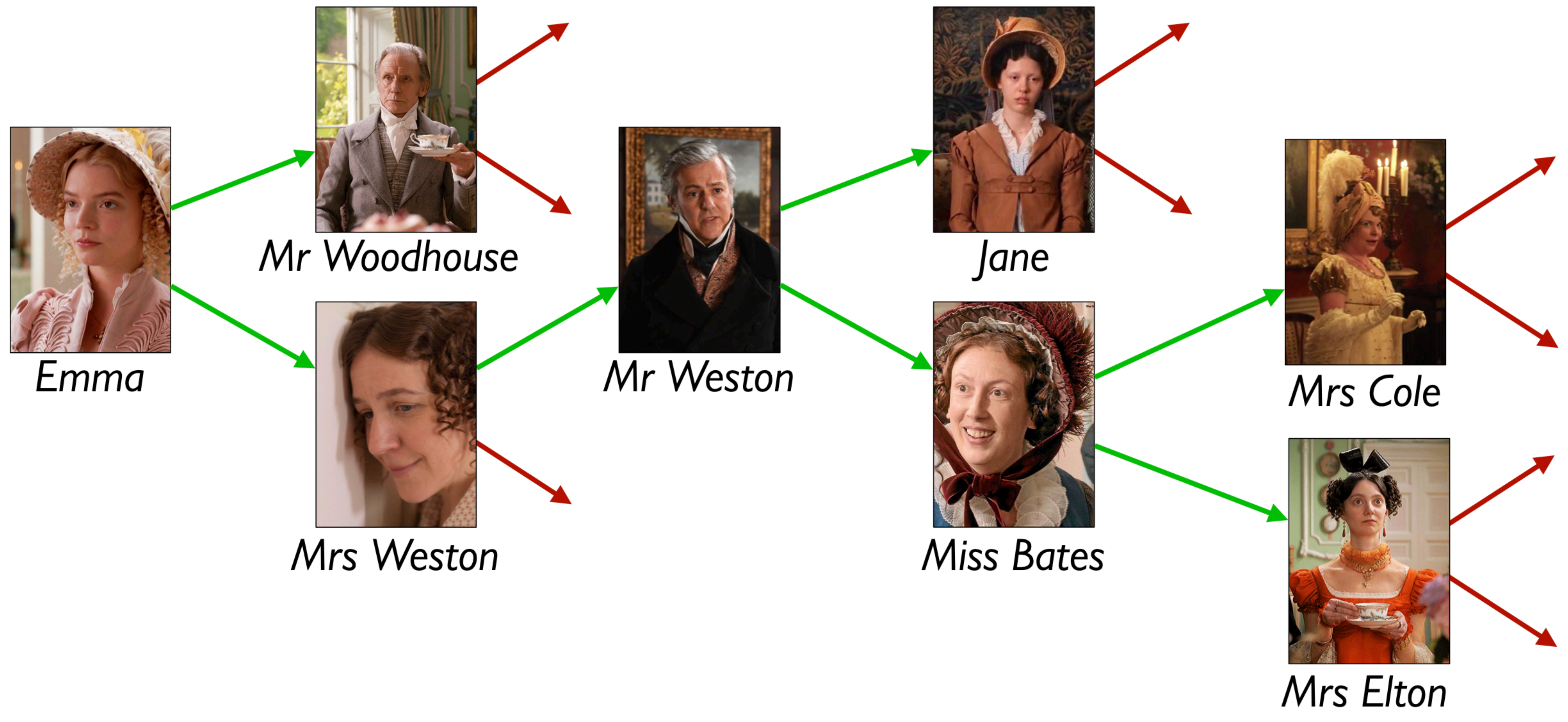
Representing rumor mills



We could represent these relations with a table, e.g.,

name :: String	***next1 :: String***	***next2 :: String***
`"Emma"`	`"Mr Woodhouse"`	`"Mrs Weston"`
`"Mr Woodhouse"`		
…	…	…

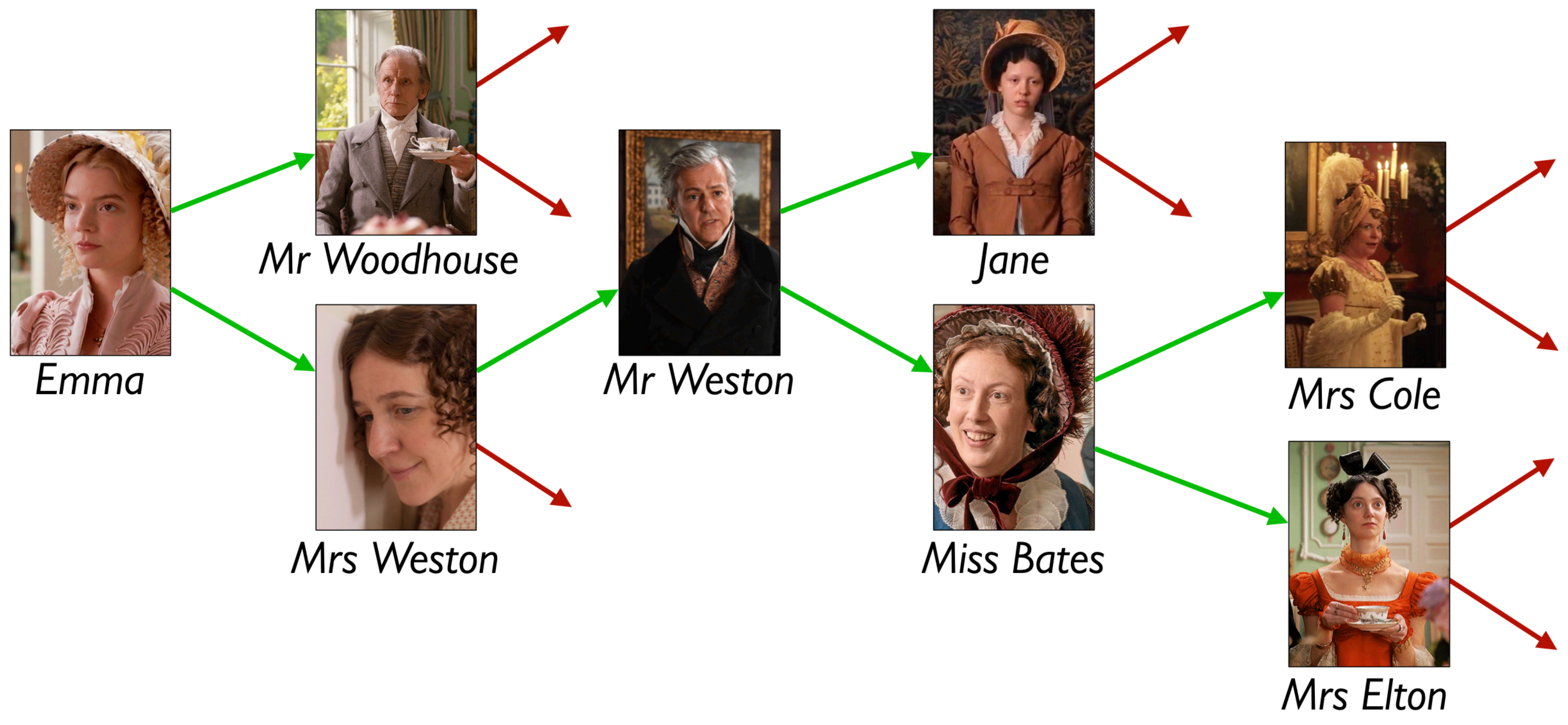
Representing rumor mills



Using a table doesn't give us any straightforward way to process the rumor mill.

Could we use something *like* a list but representing the relations?

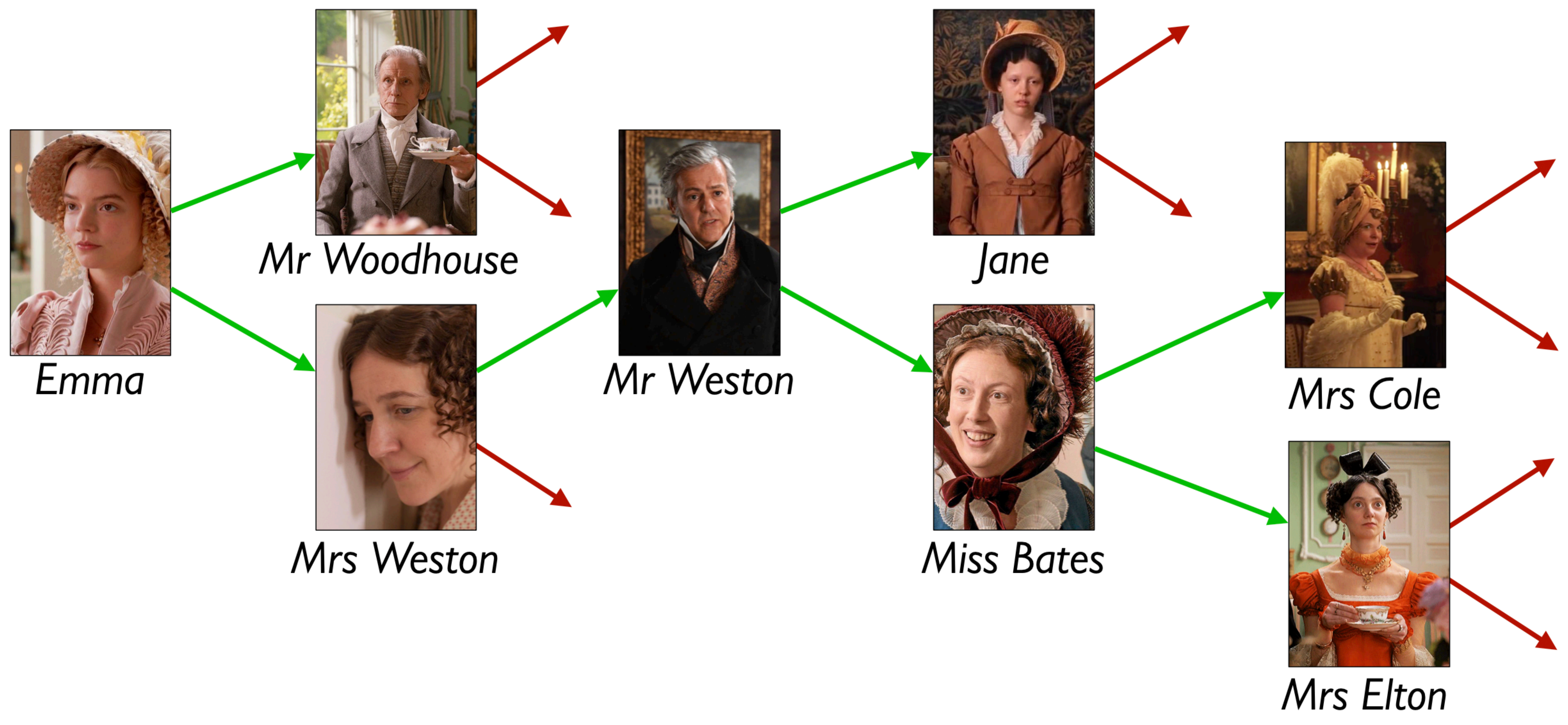
Representing rumor mills



```
data Person:
  | person(name :: String, next1 :: Person, next2 :: Person)
end
```

How about this?

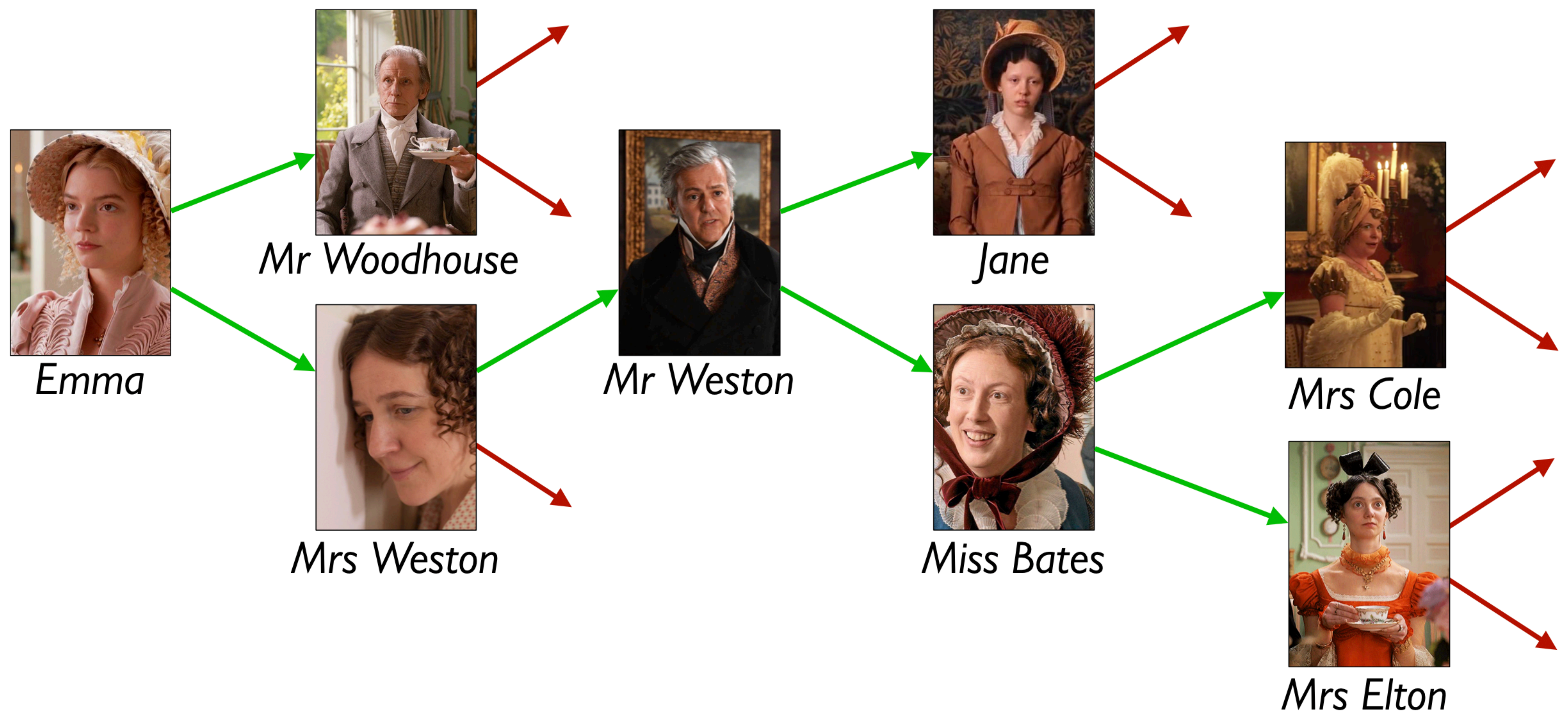
Representing rumor mills



```
data Person:
  | person(name :: String, next1 :: Person, next2 :: Person)
end
```

Some people don't gossip to anyone else – the red arrows above.

Representing rumor mills



```
data RumorMill:
  | no-one
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end
```

How about this?

Example rumor mills

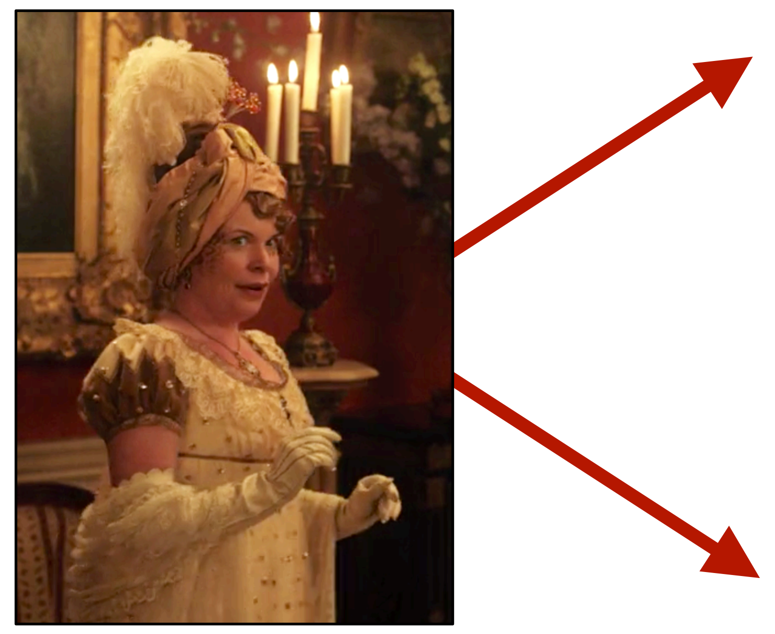
```
data RumorMill:
  | no-one
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end
```

```
no-one
```

Example rumor mills

```
data RumorMill:
  | no-one
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end
```

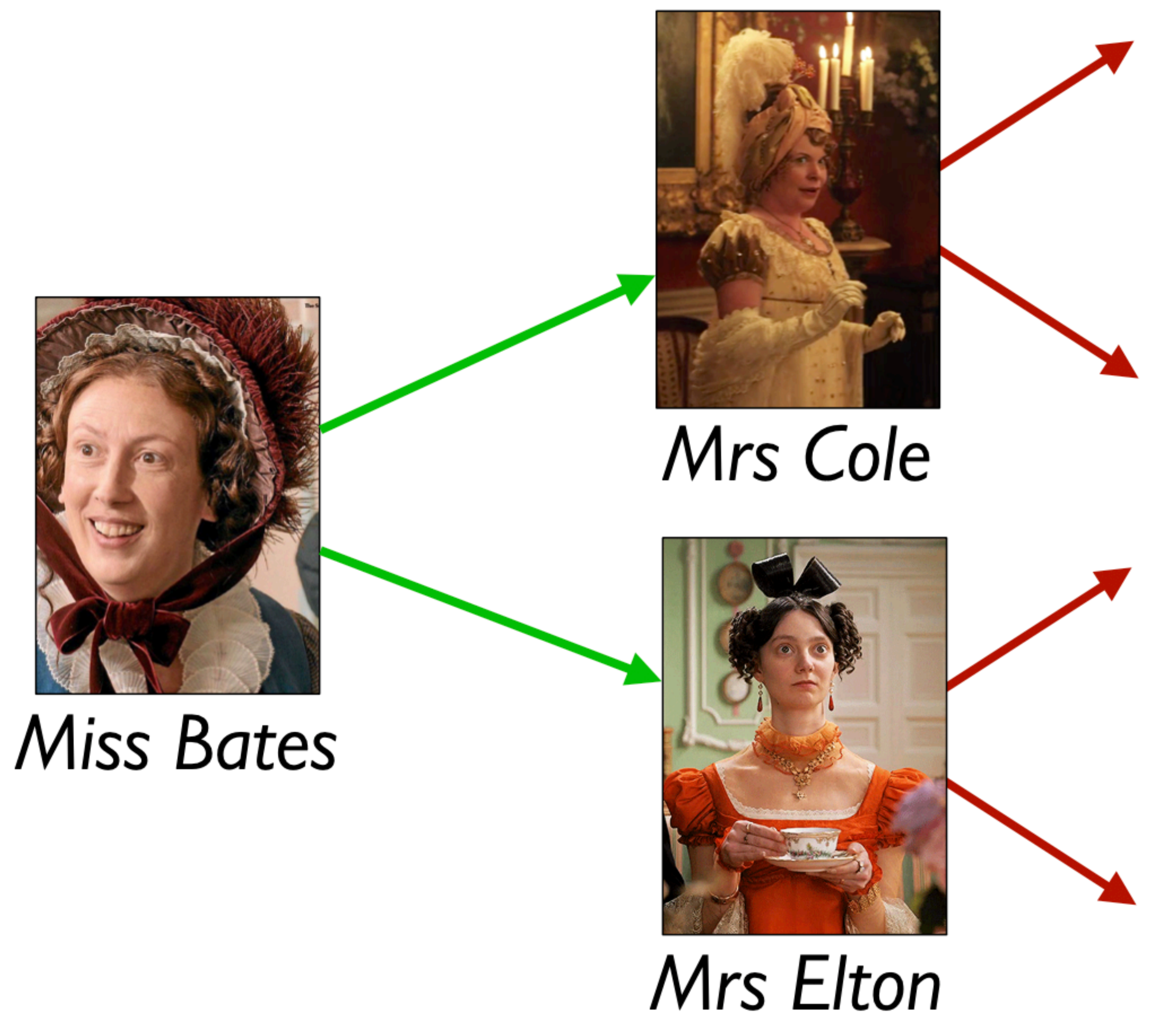
```
gossip("Mrs Cole", no-one, no-one)
```



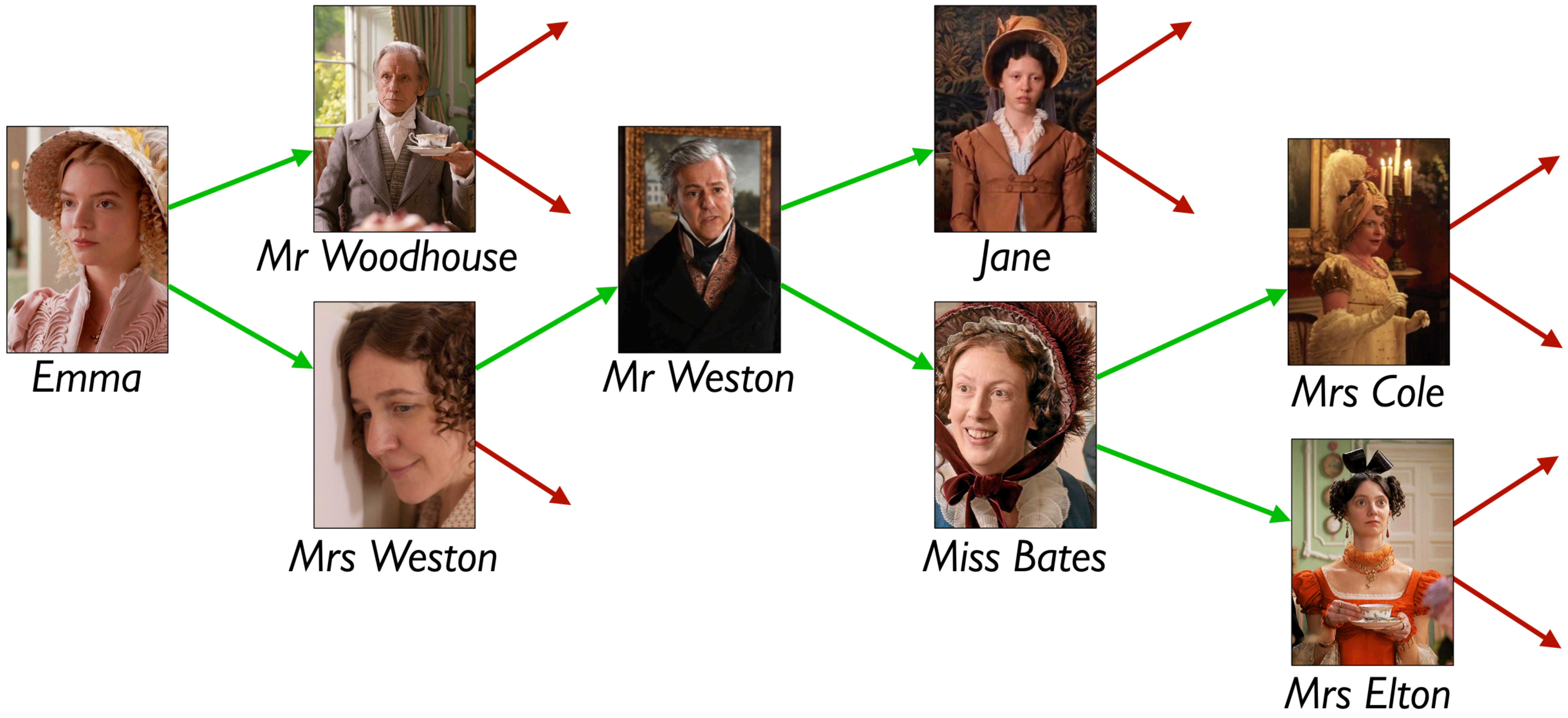
Example rumor mills

```
data RumorMill:
  | no-one
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end
```

```
gossip("Miss Bates",
  gossip("Mrs Cole", no-one, no-one)
  gossip("Mrs Elton", no-one, no-one))
```



```
gossip("Emma",
  gossip("Mr Woodhouse", no–one, no–one),
  gossip("Mrs Weston",
    gossip("Mr Weston",
      gossip("Jane", no–one, no–one),
      gossip("Miss Bates",
        gossip("Mrs Cole", no–one, no–one),
        gossip("Mrs Elton", no–one, no–one))),
    no–one))
```



Example using names for parts:

```
MRS-COLE-MILL = gossip("Mrs Cole", no-one, no-one)

MRS-ELTON-MILL = gossip("Mrs Elton", no-one, no-one)

MISS-BATES-MILL = gossip("Miss Bates", MRS-COLE-MILL, MRS-ELTON-MILL)

JANE-MILL = gossip("Jane", no-one, no-one)

MR-WESTON-MILL = gossip("Mr Weston", JANE-MILL, MISS-BATES-MILL)

MRS-WESTON-MILL = gossip("Mrs Weston", MR-WESTON-MILL, no-one)

MR-WOODHOUSE-MILL = gossip("Mr Woodhouse", no-one, no-one)

EMMA-MILL = gossip("Emma", MR-WOODHOUSE-MILL, MRS-WESTON-MILL)
```

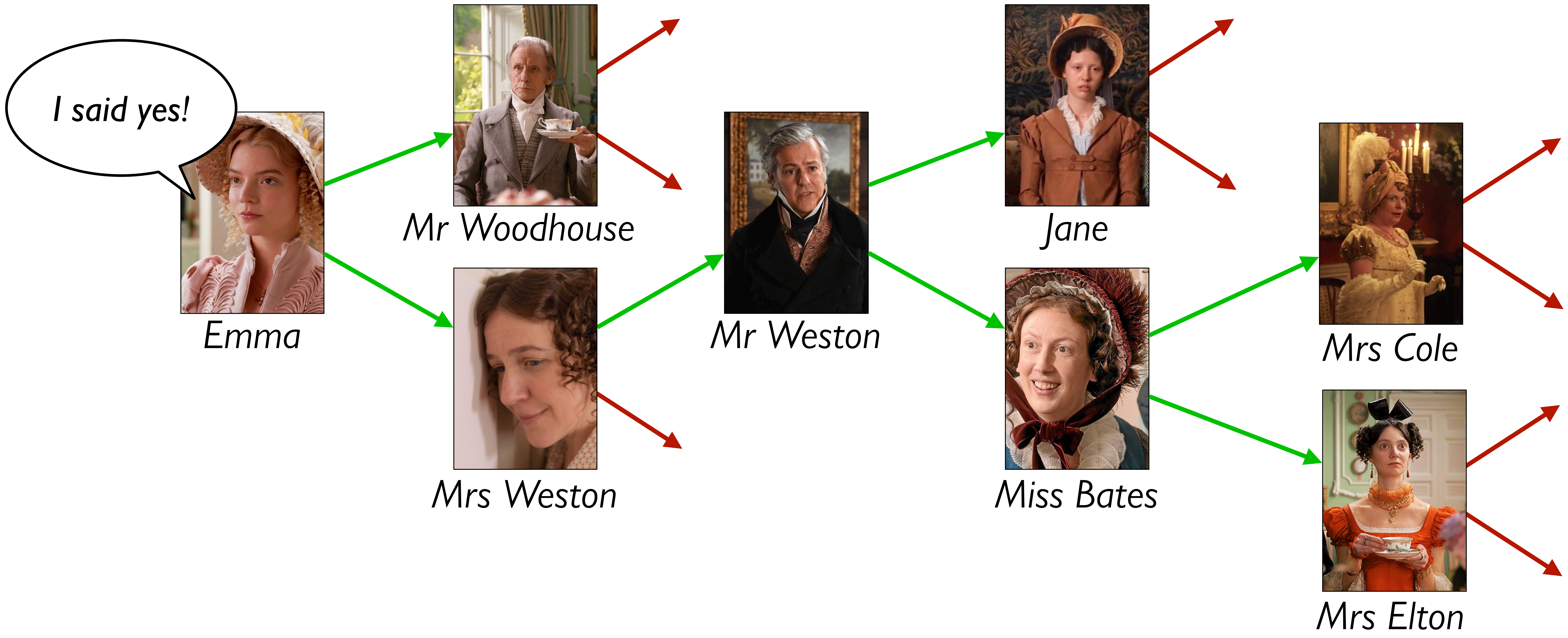
A *RumorMill* is a type of structure called a ***tree***.

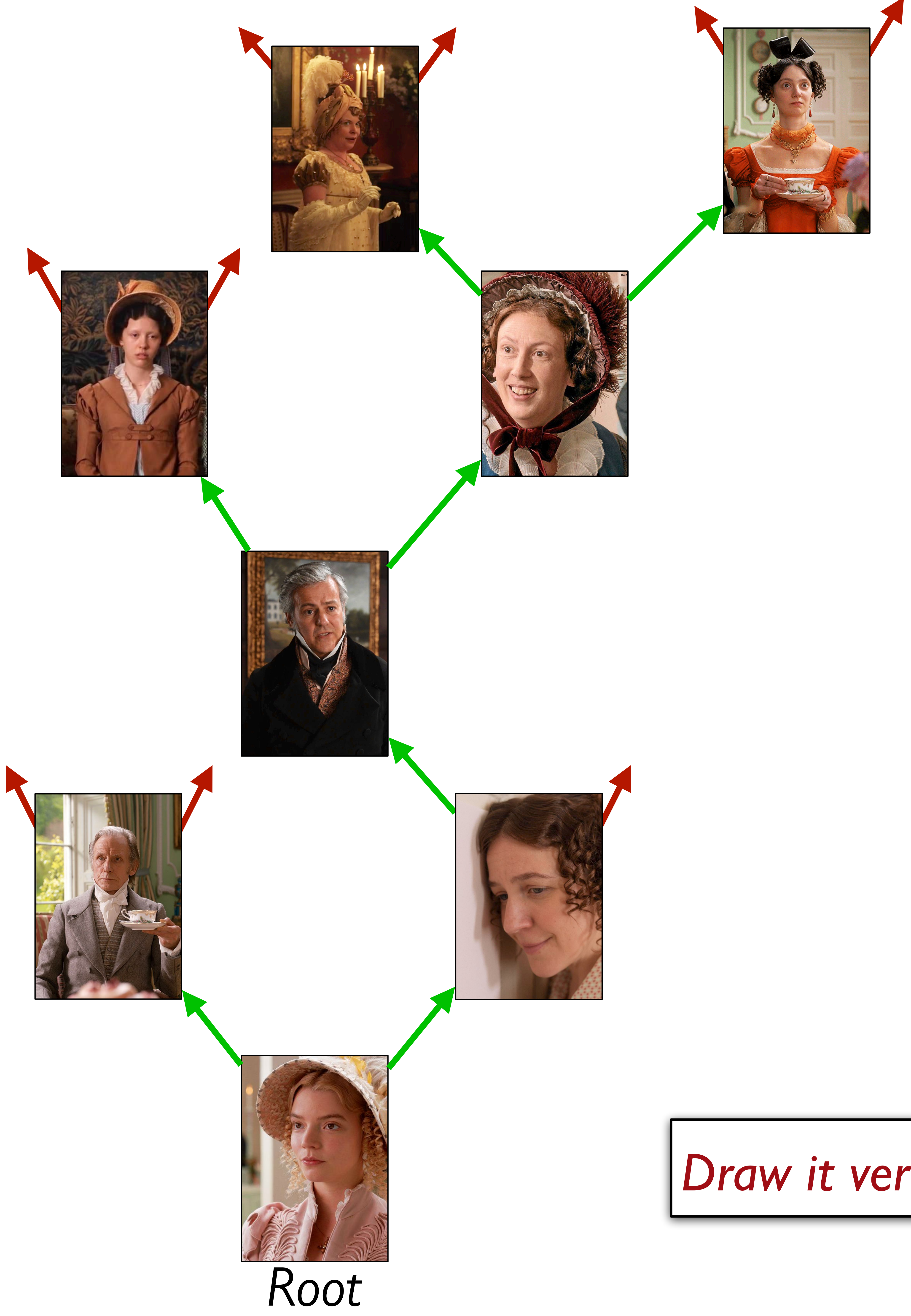
Each element in the tree is called a ***node***.

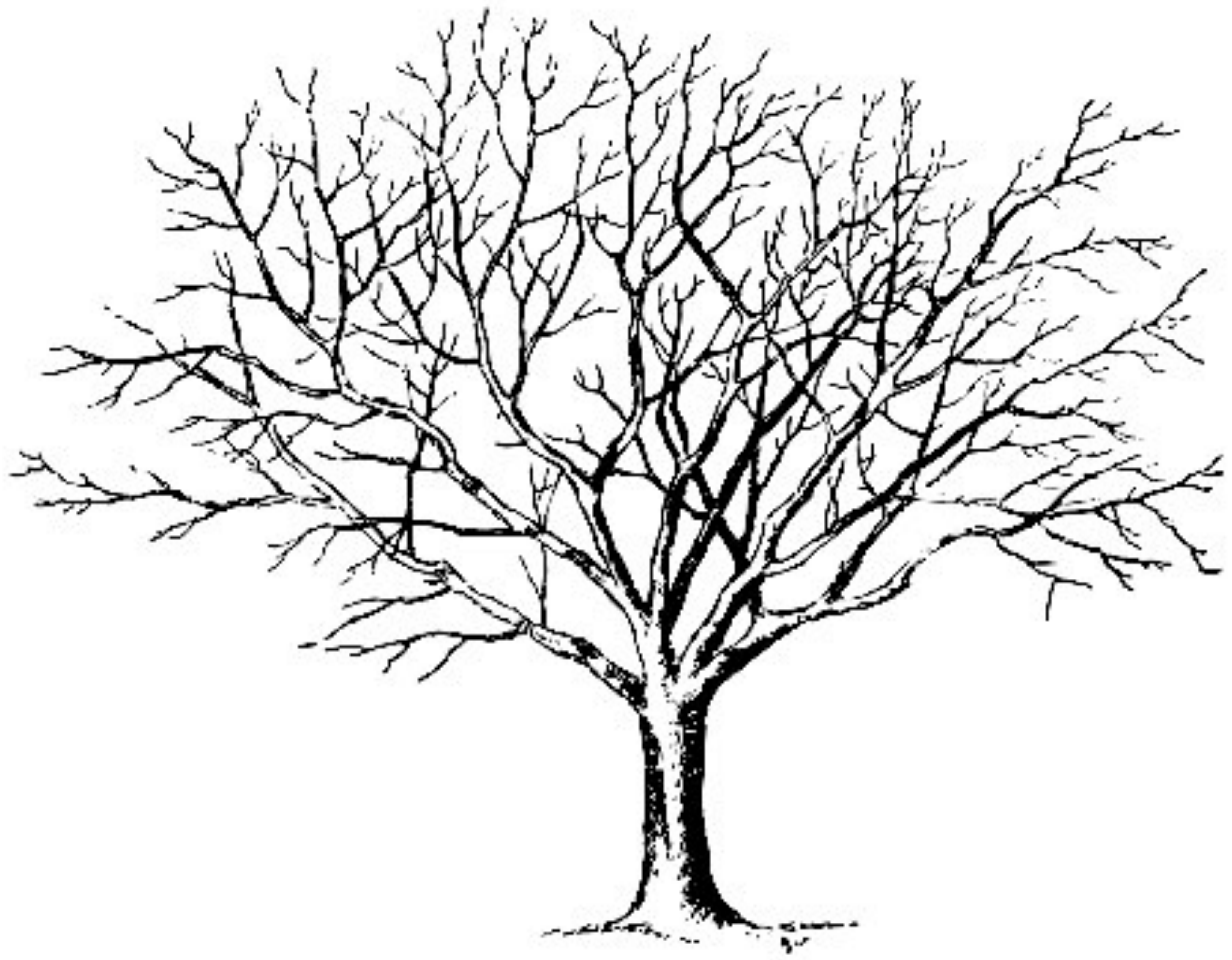
The first node in the tree is called the ***root***.

A node with no children is called a *leaf*.

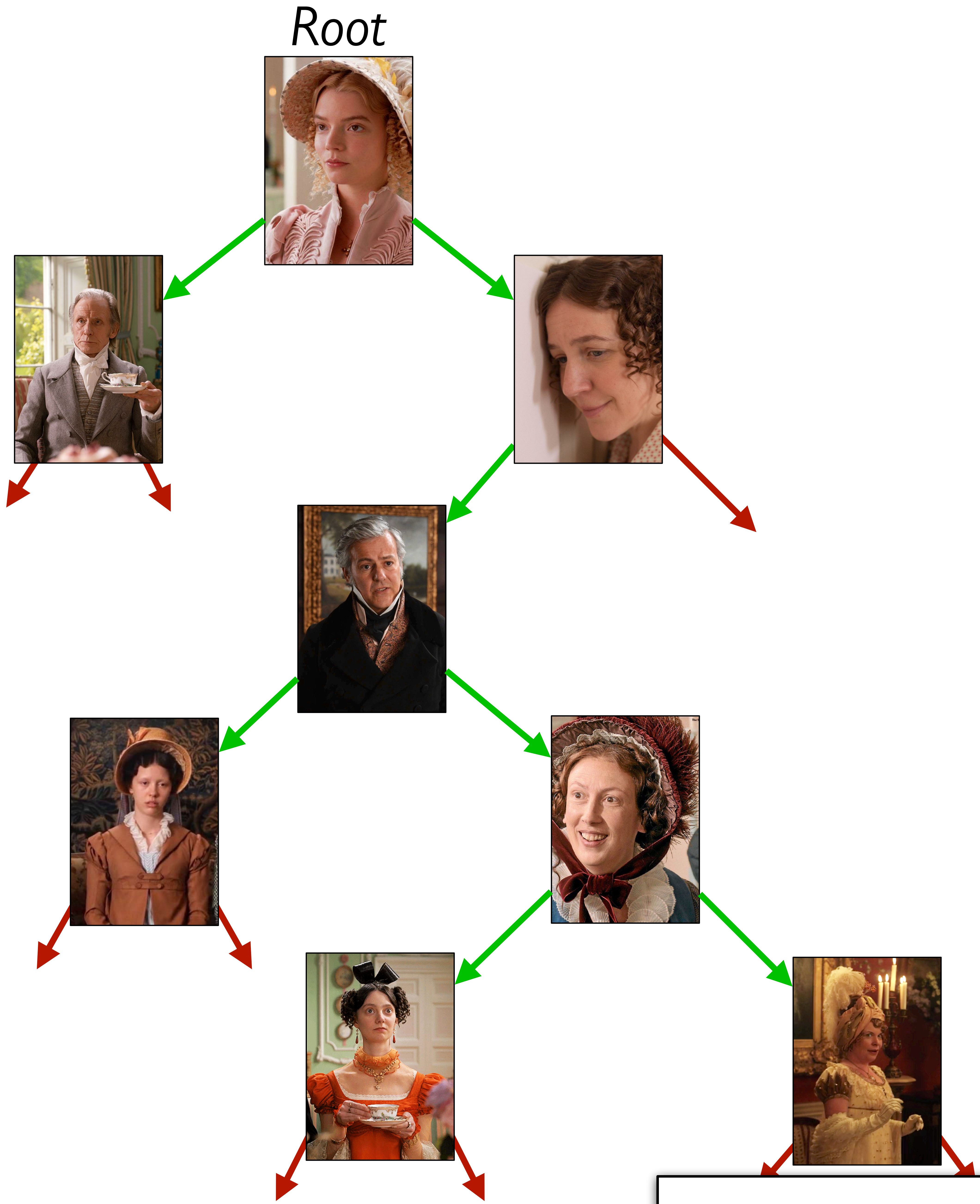
Like a list, a tree is recursive: Every subtree is a tree.

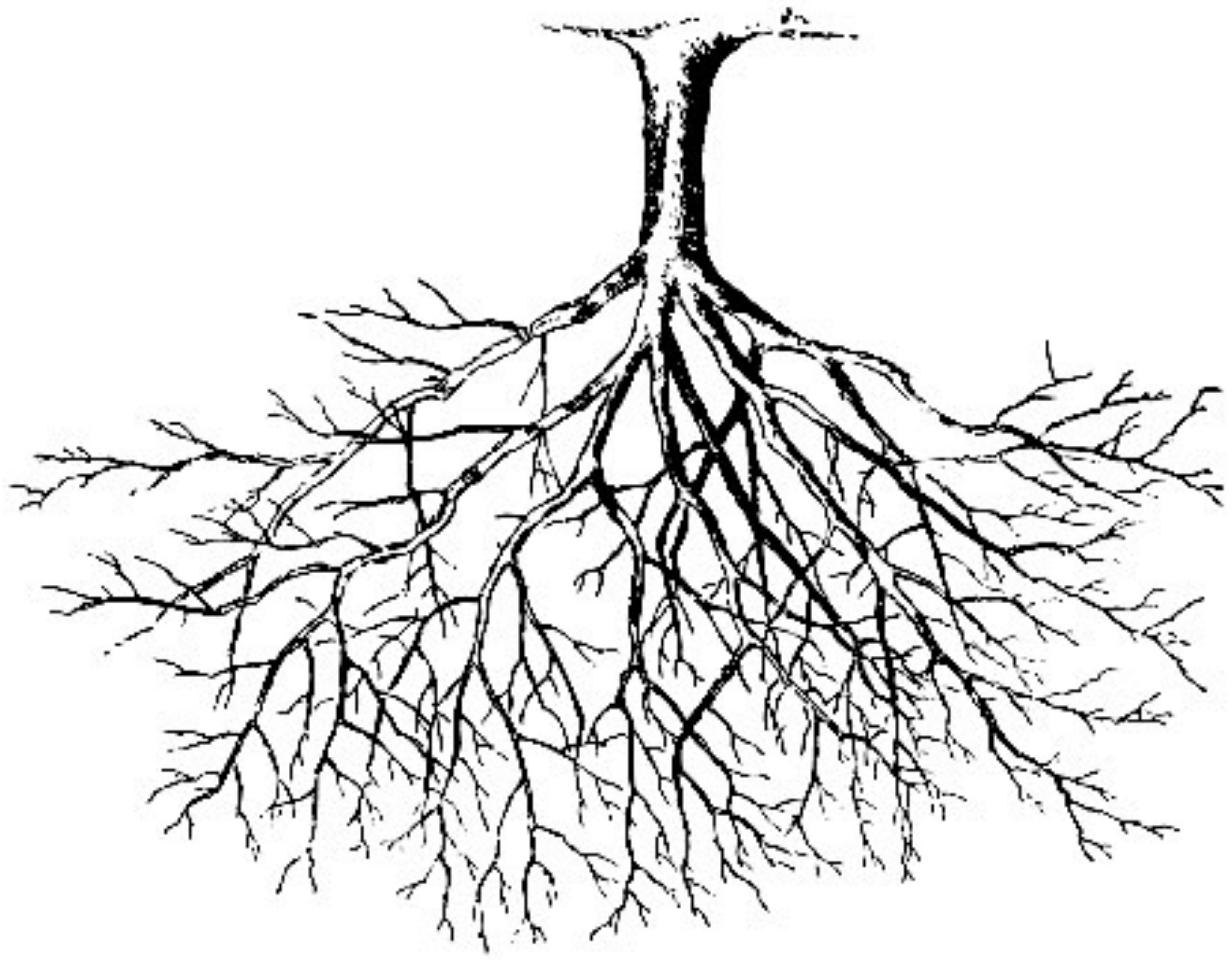

I said yes!
Emma
Mr Woodhouse
Mrs Weston
Mr Weston
Jane
Miss Bates
Mrs Cole
Mrs Elton





Draw it vertically and you can see it’s a tree!





Computer scientists are weird.

Tree

```
data RumorMill:
  | no-one
  | gossip(name :: String,
      next1 :: RumorMill,
      next2 :: RumorMill)
end
```

Subtree

Subtree

Programming with rumors

```
data RumorMill:
  | no-one
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end
```

Programming with rumors

```
data RumorMill:
  | no-one
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end
```

Self-reference × 2

For each element, there's not just one "next" element; there are two!

Programming with rumors

Self-reference × 2

```
data RumorMill:
  | no-one
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end
#|
fun rumor-mill-template(rm :: RumorMill) -> ...:
  doc: "Template for a function with a RumorMill as input"
  cases (RumorMill) rm:
    | no-one => ...
    | gossip(name, next1, next2) =>
      ... name
      ... rumor-mill-template(next1)
      ... rumor-mill-template(next2)
  end
end
|#
```

Programming with rumors

Self-reference × 2

```
data RumorMill:
  | no-one
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end
#|
fun rumor-mill-template(rm :: RumorMill) -> ...:
  doc: "Template for a function with a RumorMill as input"
  cases (RumorMill) rm:
    | no-one => ...
    | gossip(name, next1, next2) =>
      ... name
      ... rumor-mill-template(next1)
      ... rumor-mill-template(next2)
  end
end
|#
```

Natural recursion × 2

Rumor program examples

Design the function **`is-informed`** that takes a person's name and a rumor mill and determines whether the person is part of the rumor mill.

Rumor program examples

Design the function **`gossip-length`** that takes a rumor mill and determines the length of the longest sequence of people transmitting the rumor.

Rumor program examples

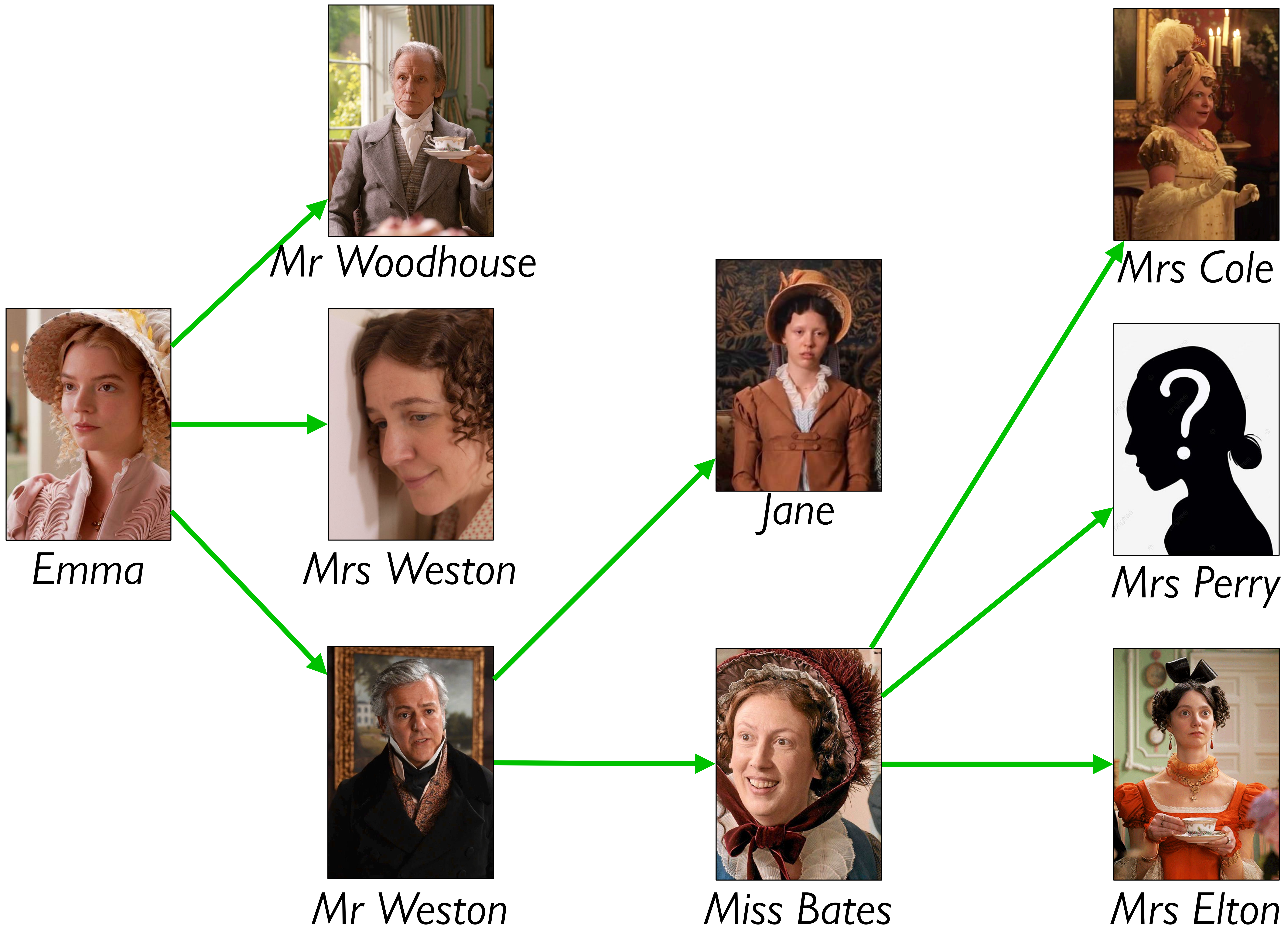
Design the function **`add-gossip`** that takes a rumor mill and two names – one new and one old – and adds the new person to the rumor mill, receiving rumors from the old person. (You can assume the old person does not already have two next persons!)

A more realistic rumor mill

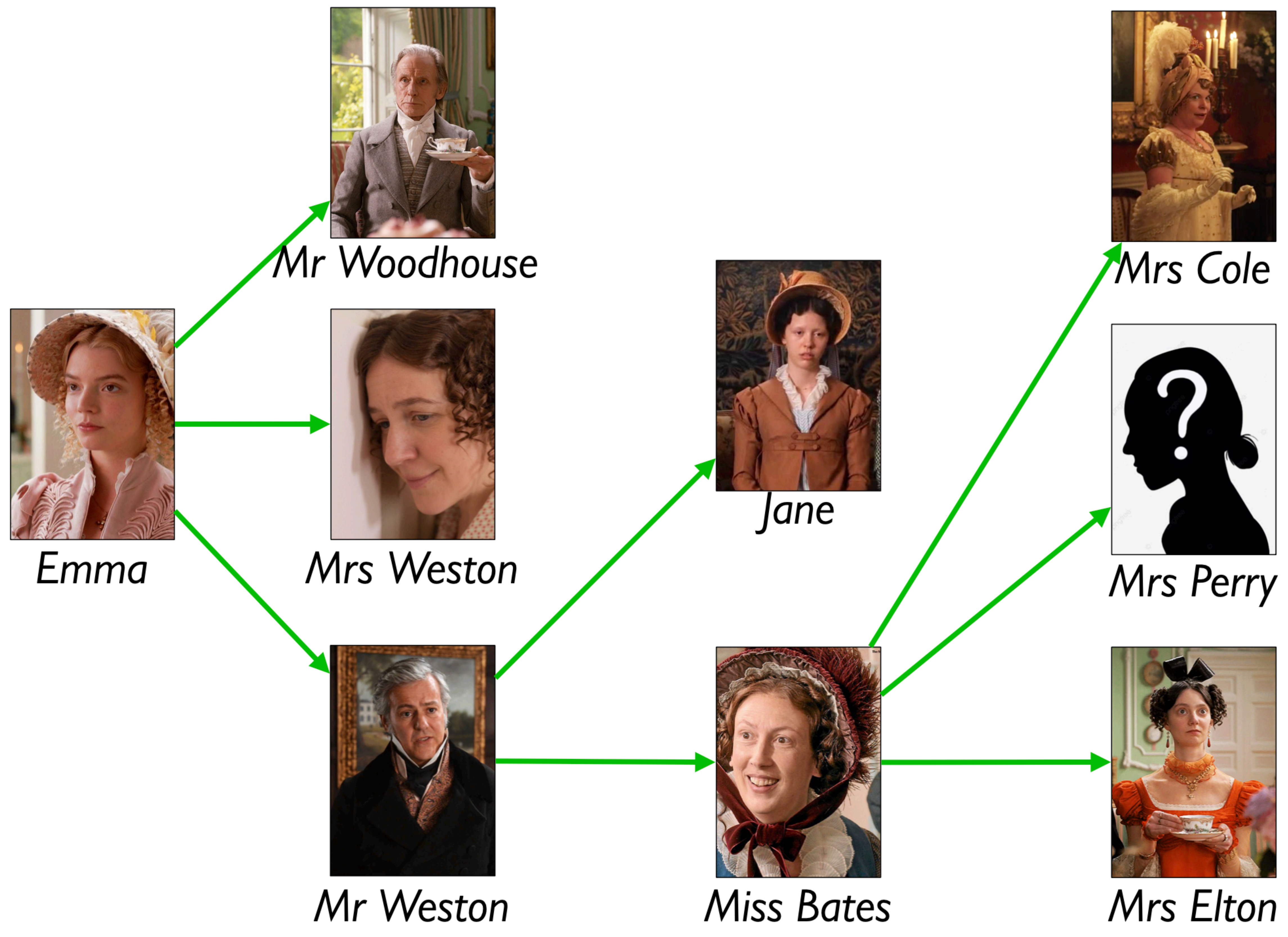
In our rumor mill, we restricted each person to spread gossip to at most two other people.

This isn’t very realistic; some gossips talk to lots of people!

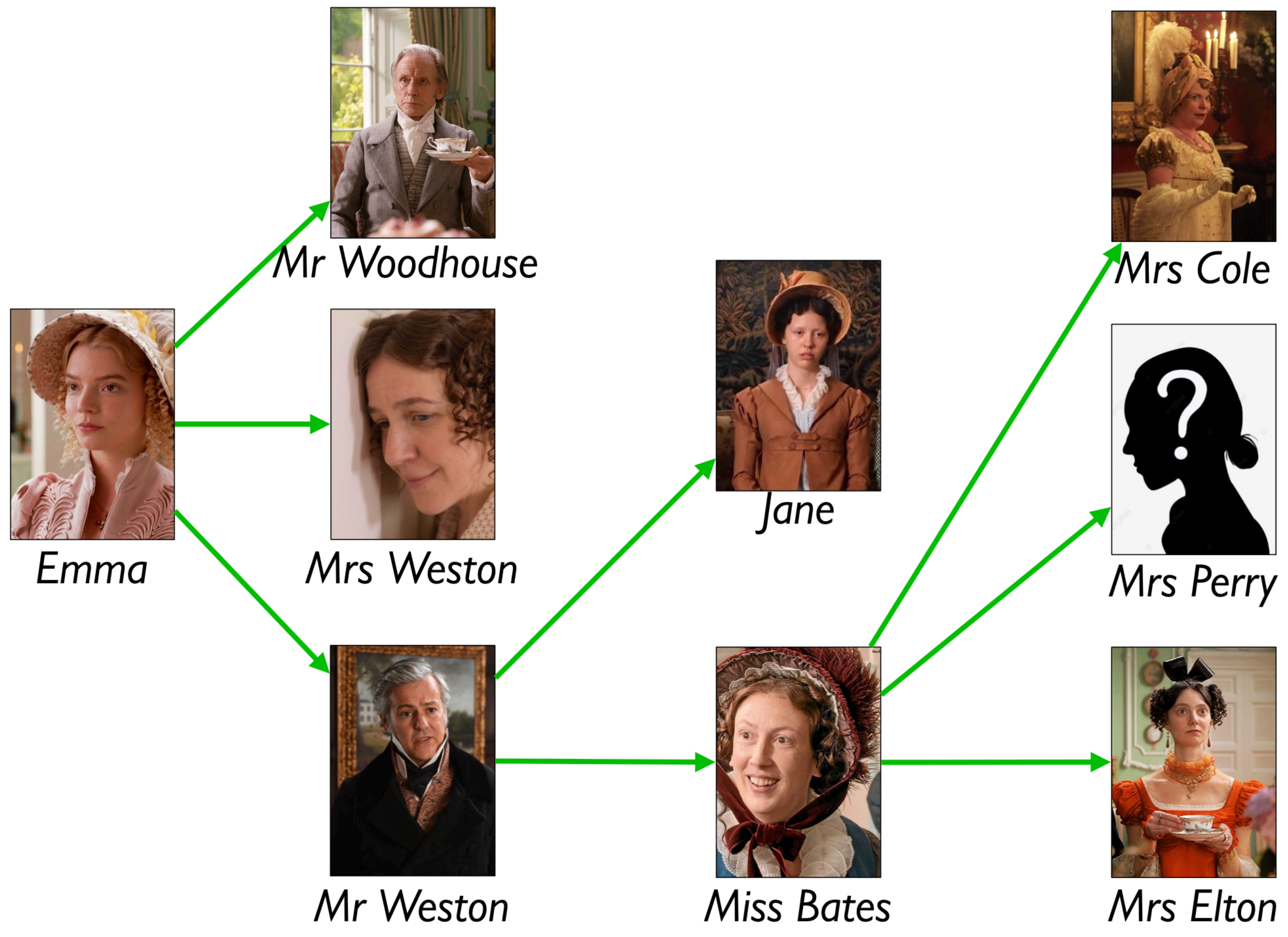
Let each gossip talk to any number of people:



How do we represent an arbitrary number of gossip connections?



How do we represent an arbitrary number of gossip connections?



```
data Gossip:
  | gossip(name :: String, next :: List<Gossip>)
end
```

```
data Gossip:
  | gossip(name :: String, next :: List<Gossip>)
end

#|
fun gossip-template(g :: Gossip) -> ...:
  ... g.name
  ... log-template(g.next)
end

fun log-template(l :: List<Gossip>) -> ...:
  cases (List) l:
    | empty => ...
    | link(f, r) =>
      ... gossip-template(f)
      ... log-template(r)
  end
end
|#
```

Design **`count-gossips`** which takes a gossip and returns the number of people informed by the gossip (including the starting person).

Acknowledgments

This lecture incorporates material from:

Jane Austen, *Emma*

Tom Ellman, Vassar College

Marc Smith, Vassar College