

# Kempe's graph-coloring algorithm

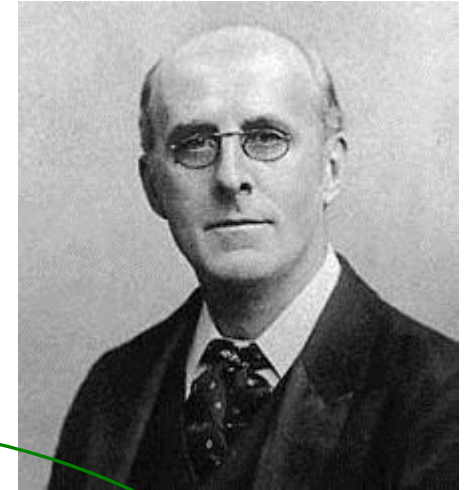
Andrew W. Appel Princeton University, 2016

These slides help explain Color.v, the graph-coloring chapter of *Verified Functional Algorithms*, a volume in the *Software Foundations* series.

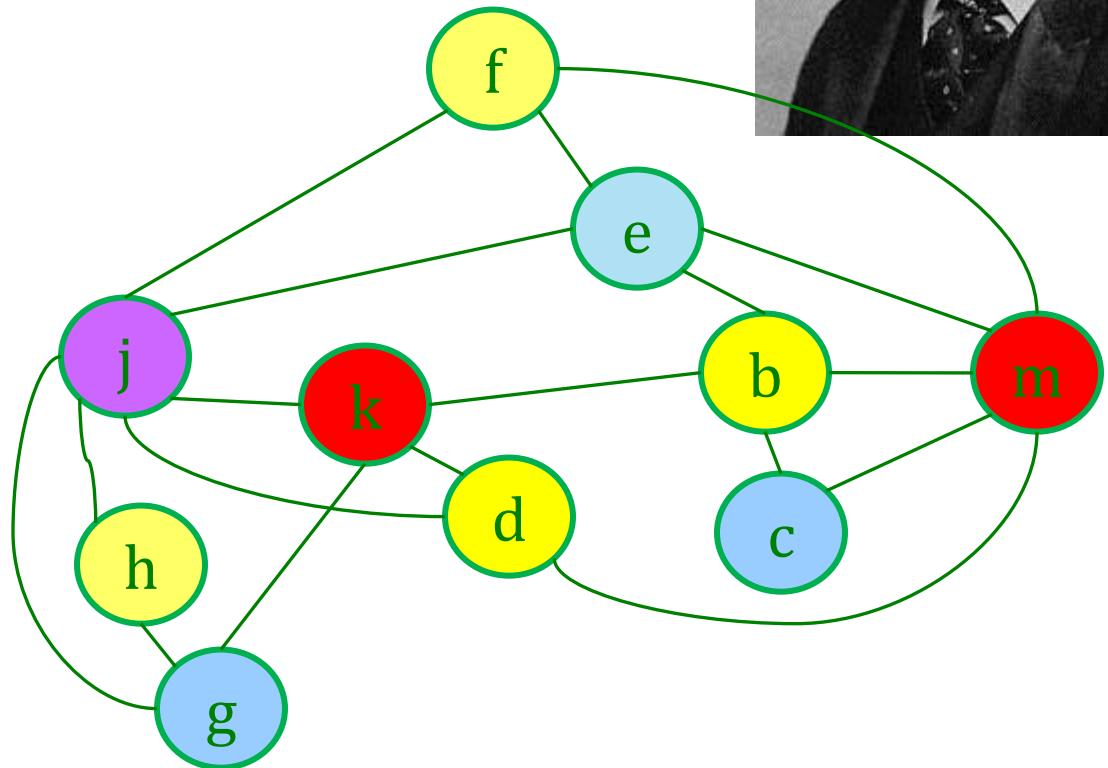
These slides are best viewed in your PDF viewer in whole-page (page-at-a-time) mode, not scrolling mode.

# Alfred B. Kempe, 1849-1922

In 1879, tried to prove the 4-color theorem:  
every planar graph can be colored using at  
most 4 colors.



That is, any nodes  
connected by an edge  
must have different  
colors.

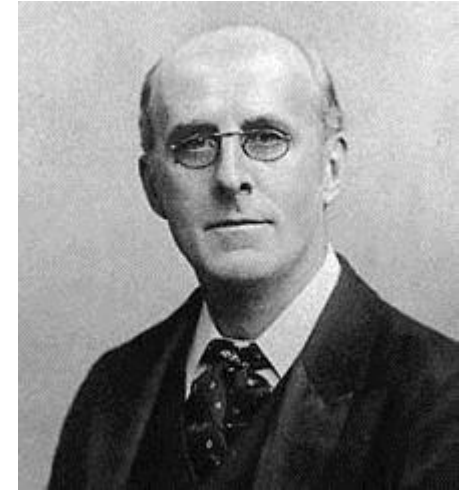


# Alfred B. Kempe, 1849-1922

In 1879, tried to prove the 4-color theorem: every planar graph can be colored using at most 4 colors.

**Failed: his proof had a bug.**

But in the process, proved the 5-color theorem: every planar graph can be colored using at most 5 colors. For use in this proof, he invented an algorithm for graph coloring that is still relevant today, for use in many applications such as register allocation in compilers.

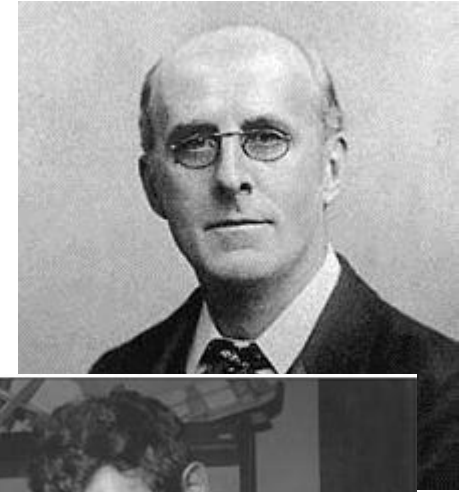


# Alfred B. Kempe, 1849-1922

In 1879, tried to prove the 4-color theorem: every planar graph can be colored using at most 4 colors.

**Failed: his proof had a bug.**

Some other guys fixed up Kempe's buggy proof in 1976, using computers: they proved the 4-color theorem. But their proof doesn't have applications to compilers, as far as I know.

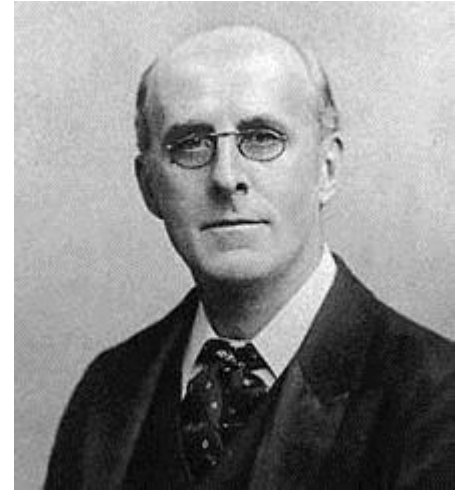


*Kenneth Appel and Wolfgang Haken in the 1970s*

# Kempe's graph-coloring algorithm

To 6-color a planar graph:

1. Every planar graph has at least one vertex of degree  $\leq 5$ .
2. Remove this vertex.
3. Color the rest of the graph with a recursive call to Kempe's algorithm.
4. Put the vertex back. It is adjacent to at most 5 vertices, which use up at most 5 colors from your "palette." Use the 6<sup>th</sup> color for this vertex.



# From 6-coloring to 5-coloring

That was Kempe's simplest algorithm, to 6-color a planar graph; or in general, to  $K$ -color a graph in class  $C$ , such that (1) every graph in class  $C$  has a node of degree  $< K$ , and (2) removing a node from a graph in class  $C$  gives you another graph in class  $C$ .

Kempe had two more algorithms:

5-color a planar graph

4-color a planar graph (but this algorithm had a bug)

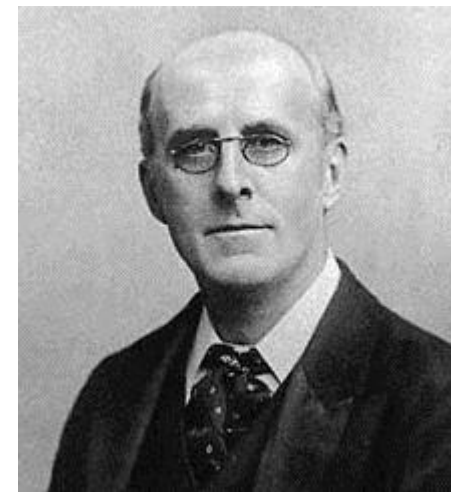
# Kempe's 5-coloring algorithm

To **5-color** a planar graph:

1. Every planar graph has at least one vertex of degree  $\leq 5$ .
2. Remove this vertex.
3. Color the rest of the graph with a recursive call to Kempe's algorithm.
4. Put the vertex back. It is adjacent to at most 5 vertices. How many different colors are used in these 5 vertices?

Four or less: use the fifth color for this vertex.

Five: use the method of "Kempe chains", which is beyond the scope of this discussion.

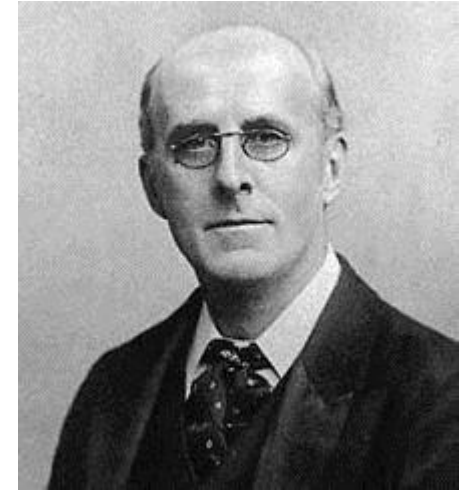


# Kempe's 5-coloring algorithm

To **5-color** a planar graph:

1. Every planar graph has at least one vertex of degree at most 5.
2. Remove this vertex.
3. Color the remaining graph with 5 colors.
4. Put the removed vertex back. Most of the time, it has at most 5 neighbors, so any different colors are used in these 5 neighbors? Four or less: use the fifth color for this vertex. Five: use the method of "Kempe chains", which is beyond the scope of this discussion.

**We will set this algorithm aside, as it does not really concern us, and go back to Kempe's simpler algorithm**





# Heuristic hack of Kempe's algorithm

To **mostly K-color** a graph (whether planar or not!)

Is there a vertex of degree  $< K$  ?

If so:

- Remove this vertex.

- Color the rest of the graph with a recursive call to the algorithm.

- Put the vertex back. It is adjacent to at most  $K-1$  vertices. They use (among them) at most  $K-1$  colors. That leaves one of your colors for this vertex.

If not:

- Remove this vertex.

- Color the rest of the graph with a recursive call.

- Put the vertex back. It is adjacent to  $\geq K$  vertices. How many colors do these vertices use among them?

- If  $< K$ : there is an unused color to use for this vertex

- If  $\geq K$ : leave this vertex uncolored.

# Heuristic hack of Kempe's algorithm

To **mostly K-color** a graph (whether planar or not!)

Is there a vertex of degree  $< K$  ?

If so:

Remove this vertex.

Color the rest of the graph with a recursive call.

Put the vertex back. It is adjacent to at most  $K-1$  vertices (and these vertices use among them)

If not:

Remove this vertex.

Color the rest of the graph with a recursive call.

Put the vertex back. It is adjacent to at most  $K$  vertices (and these vertices use among them) at most  $K-1$  colors. The vertex can use the  $K$ th color.

If  $< K$ : there is an unused color to use for this vertex

If  $\geq K$ : **leave this vertex uncolored.**

What?  
Are we allowed to do that?

Yes!  
This is an algorithm to  
"mostly K-color" a graph.

# Heuristic hack of Kempe's algorithm

To **mostly K-color** a graph (whether planar or not!)

In the application of register allocation for compilers, the uncolored nodes correspond to variables that are “spilled” to memory instead of held in registers.

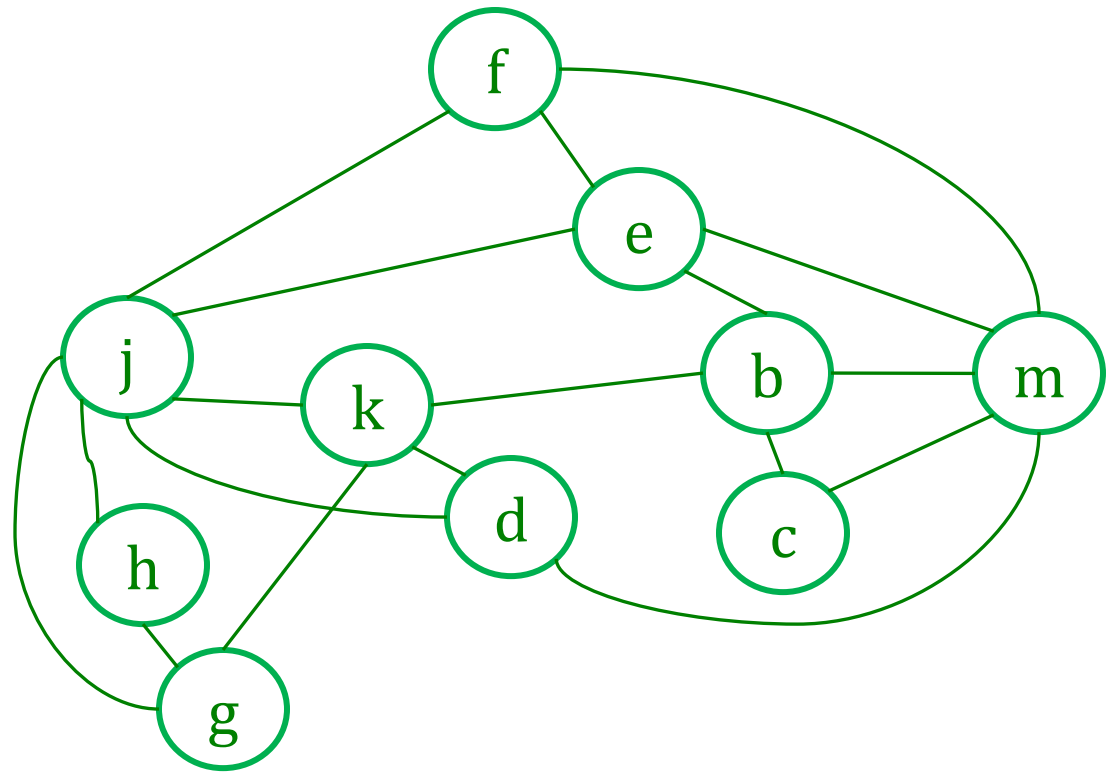
This variation of Kempe's algorithm was invented by Gregory Chaitin in 1981.



Gregory Chaitin

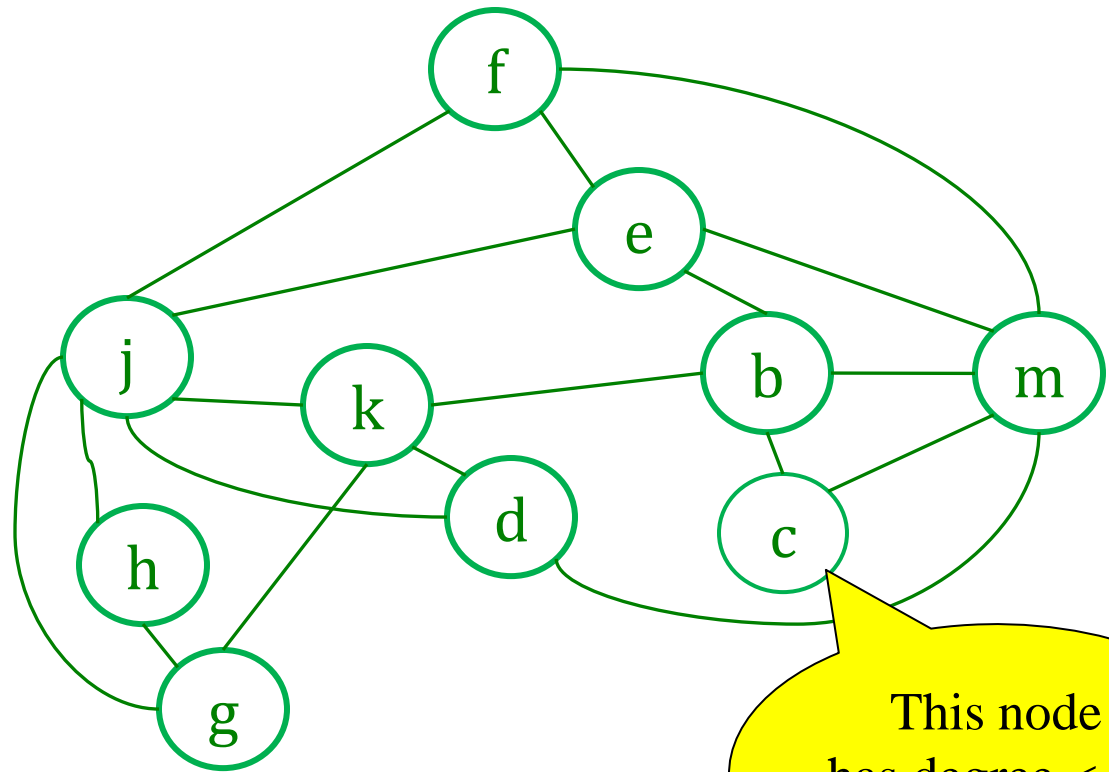
If  $\geq K$ : leave this vertex uncolored.

# Example: 3-color this graph



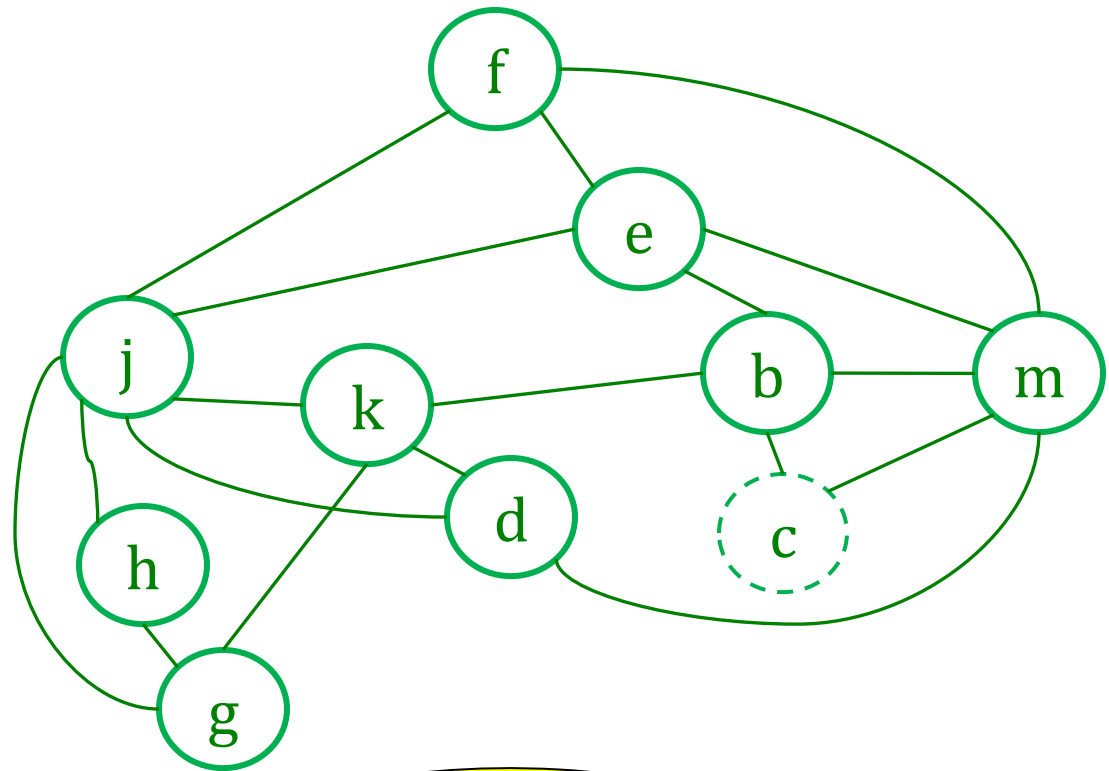
Stack:

# Example: 3-color this graph



Stack:

# Example: 3-color this graph

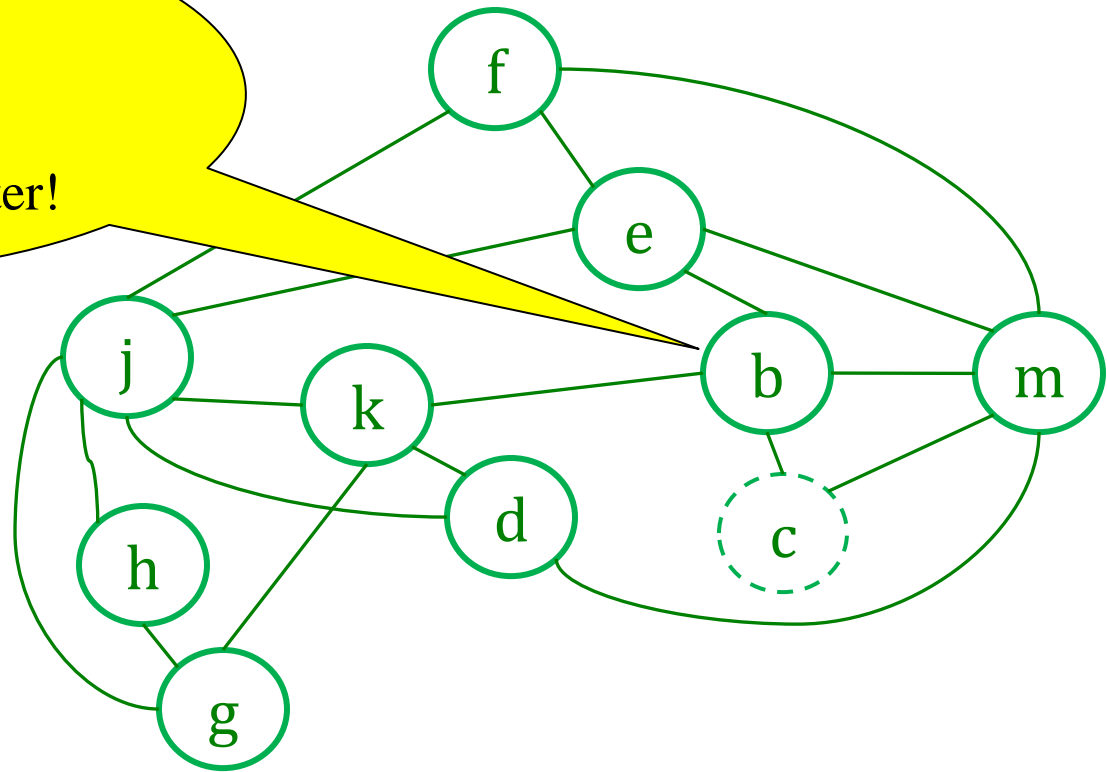


Stack: c

Push node c on  
the stack

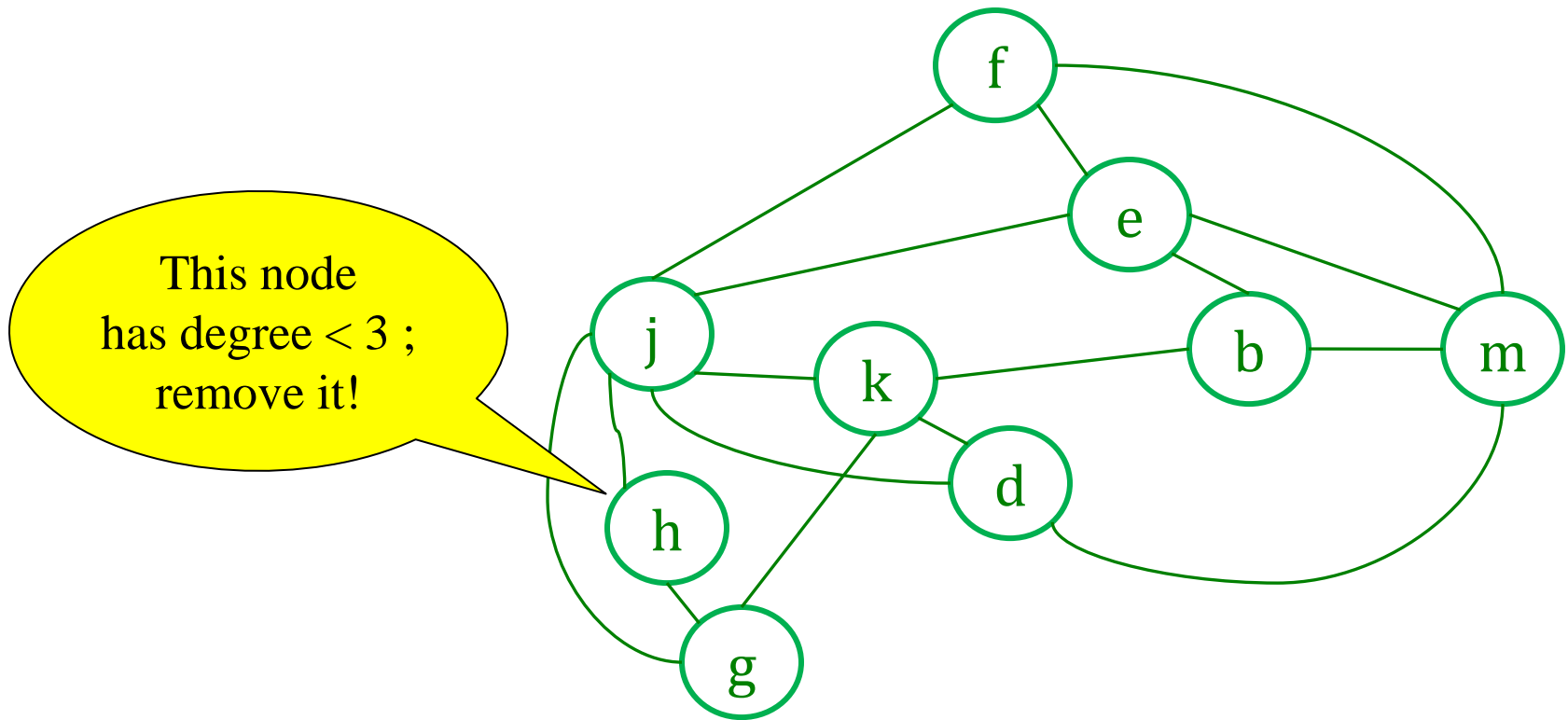
# Example: 3-color this graph

Removing  $c$   
lowers the degree  
of nodes  $b$  and  $m$ ;  
that will be helpful later!



Stack:  $c$

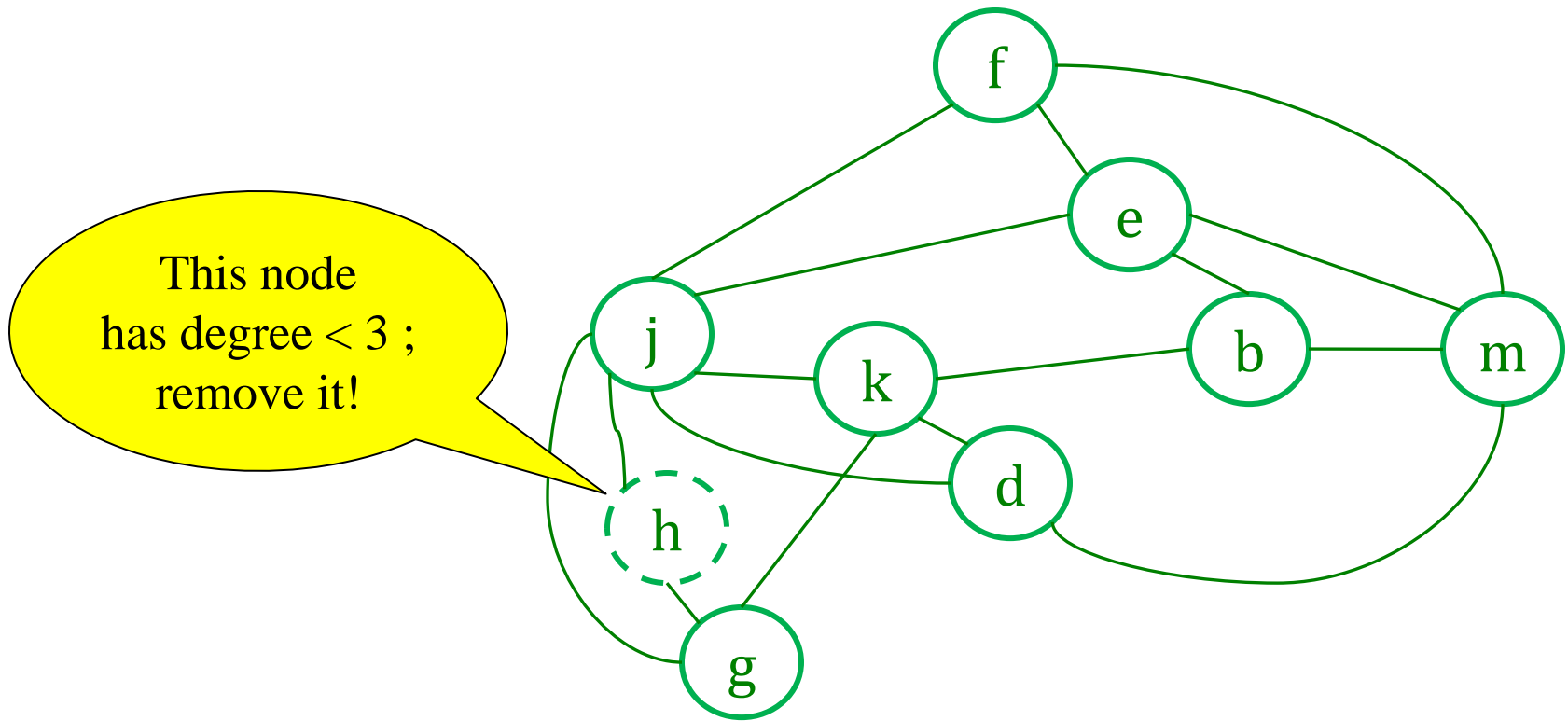
# Example: 3-color this graph



Stack: c

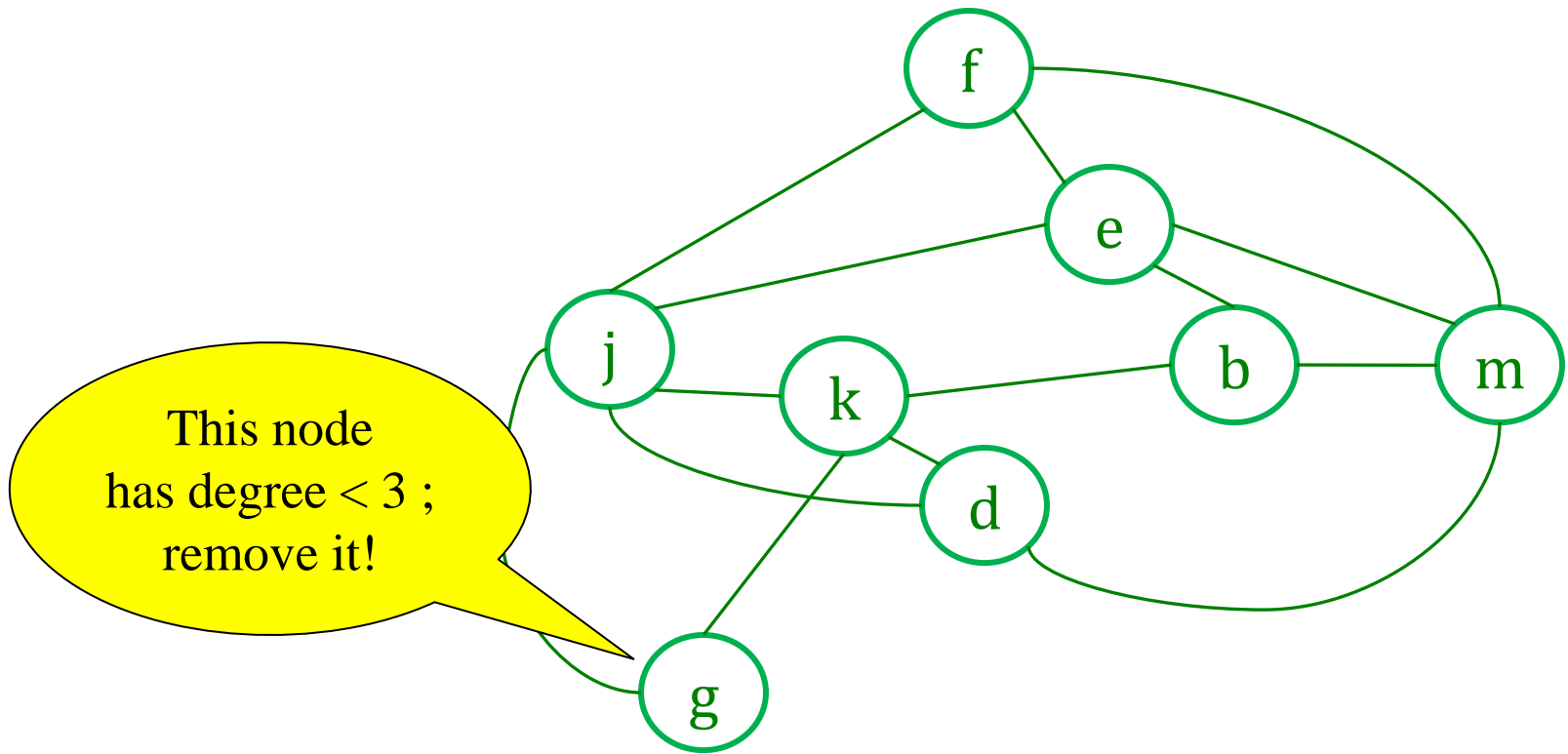


# Example: 3-color this graph



Stack: h c

# Example: 3-color this graph

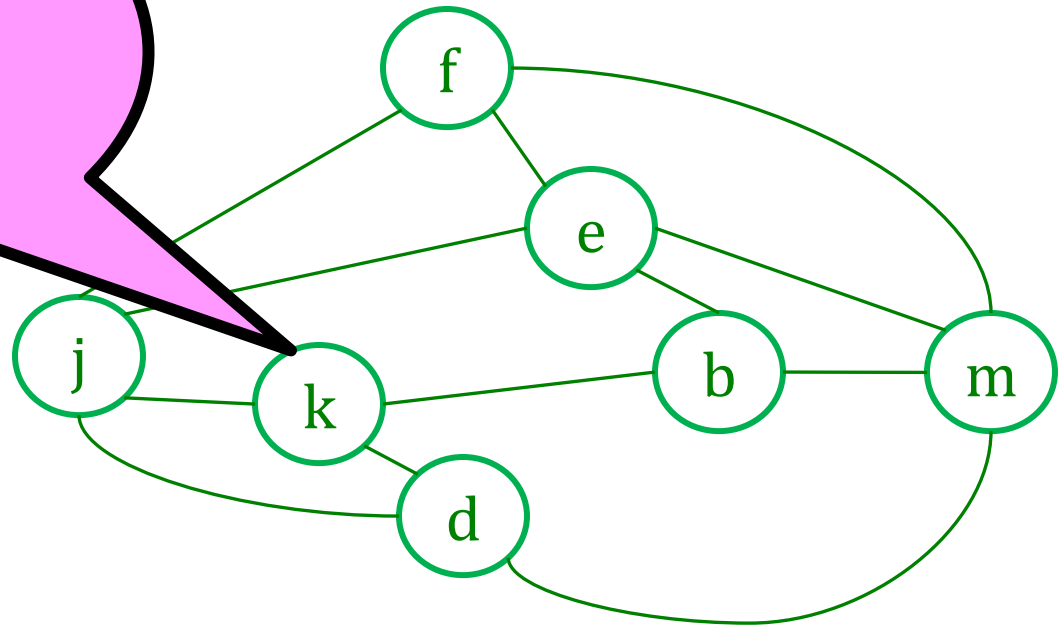


Stack: h c

# Example: 3-color this graph

No node has degree  $< 3$

Pick a node arbitrarily,  
remove it, and  
push it on the stack

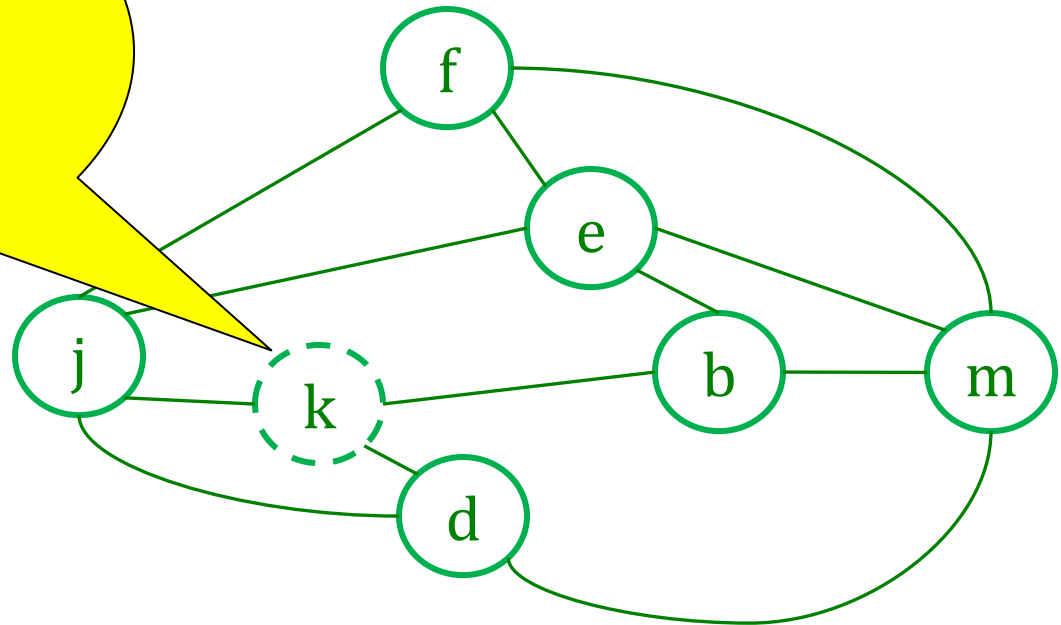


Stack: g h c

# Example: 3-color this graph

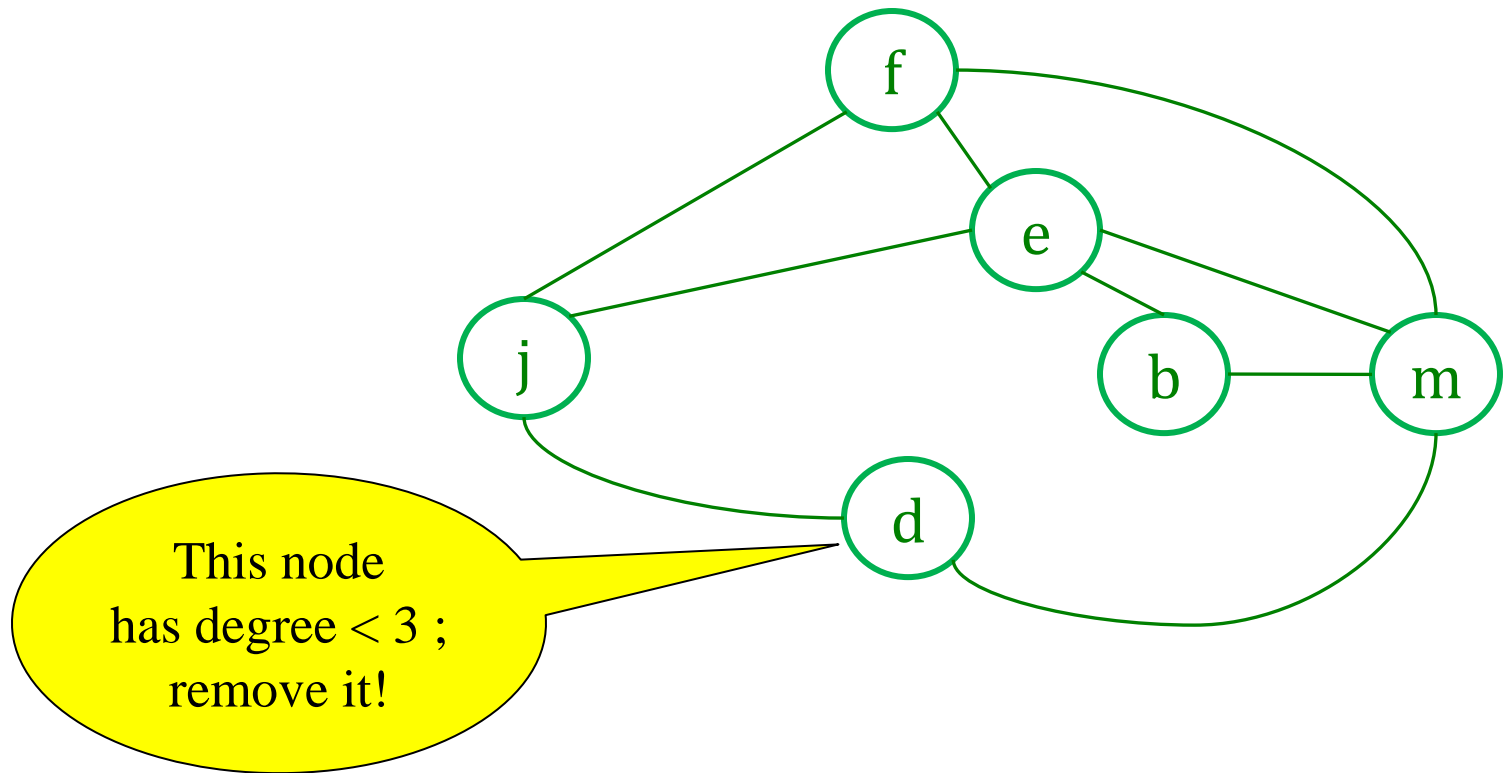
No node has degree  $< 3$

Pick a node arbitrarily,  
remove it, and  
push it on the stack



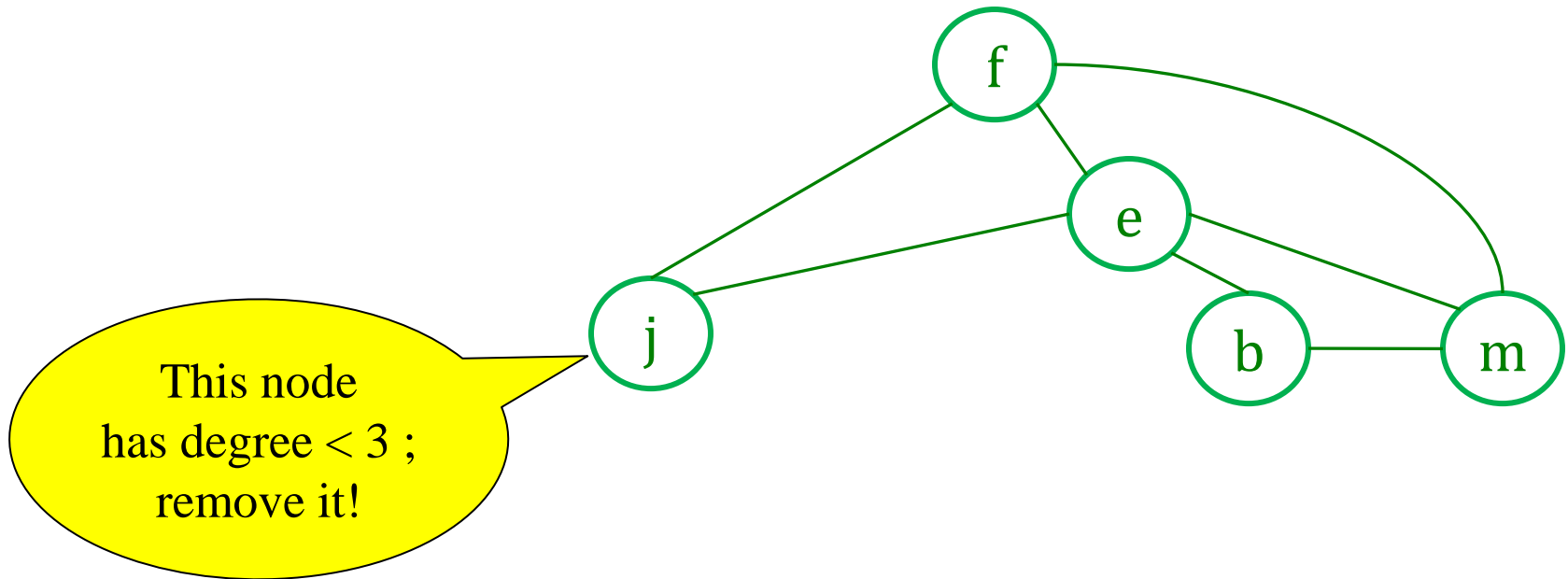
Stack: k g h c

# Example: 3-color this graph



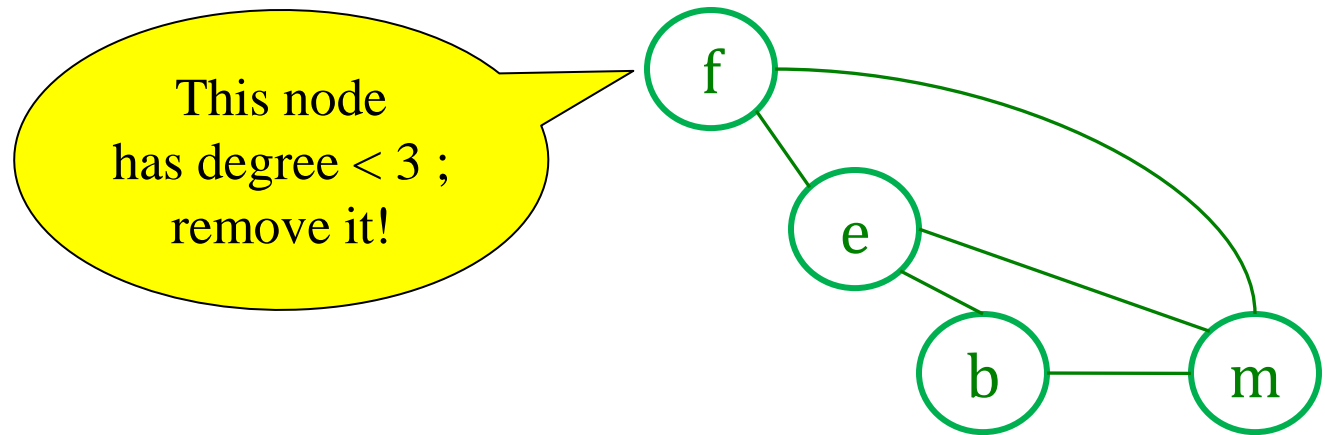
Stack: k g h c

# Example: 3-color this graph



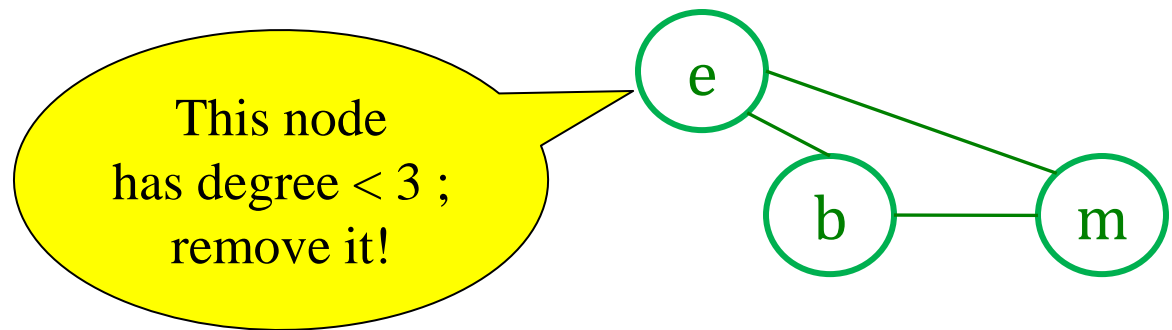
Stack: d k g h c

# Example: 3-color this graph



Stack: j d k g h c

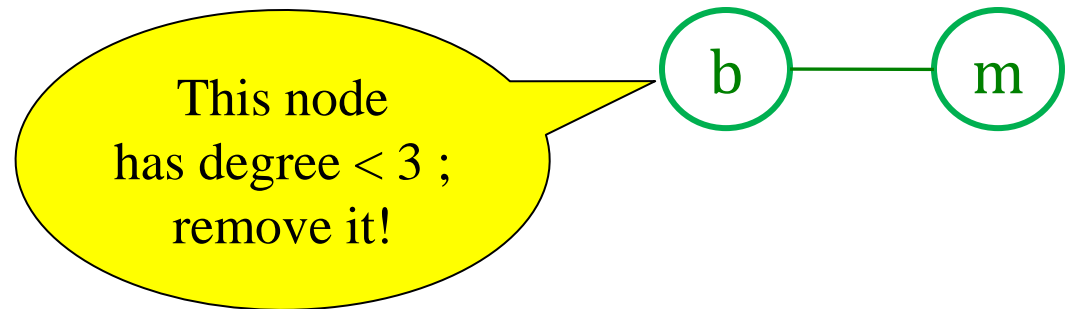
# Example: 3-color this graph



Stack: f j d k g h c

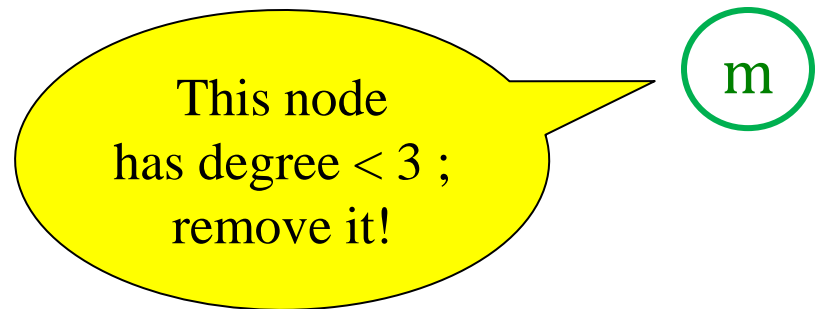


# Example: 3-color this graph



Stack: e f j d k g h c

# Example: 3-color this graph



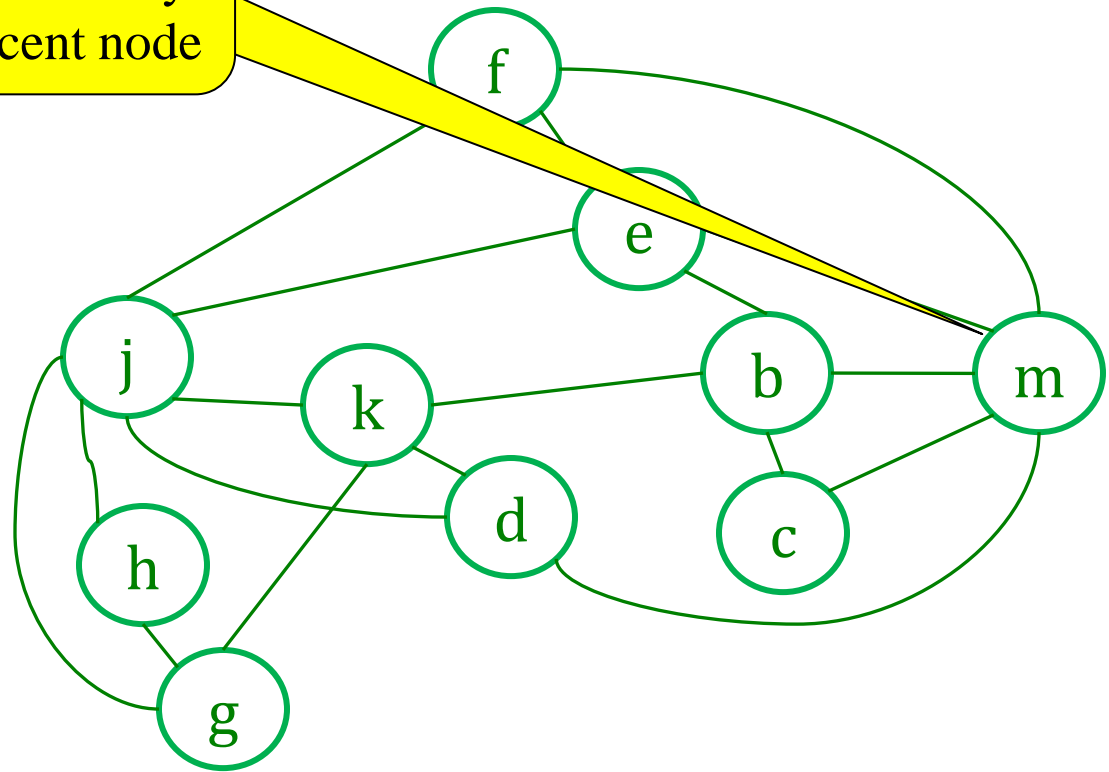
Stack: b e f j d k g h c

# Example: 3-color this graph

Stack: m b e f j d k g h c

# Now, color the nodes in stack order

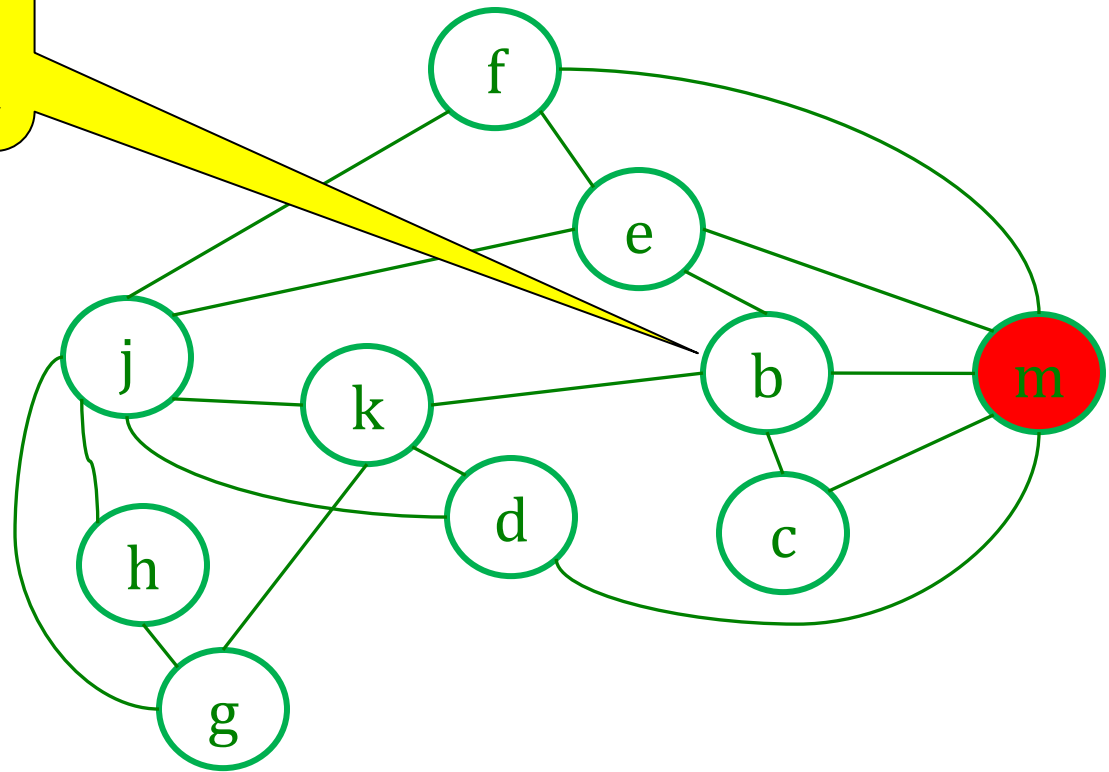
Find a color for this node that's not already used in an adjacent node



Stack: m b e f j d k g h c

# Now, color the nodes in stack order

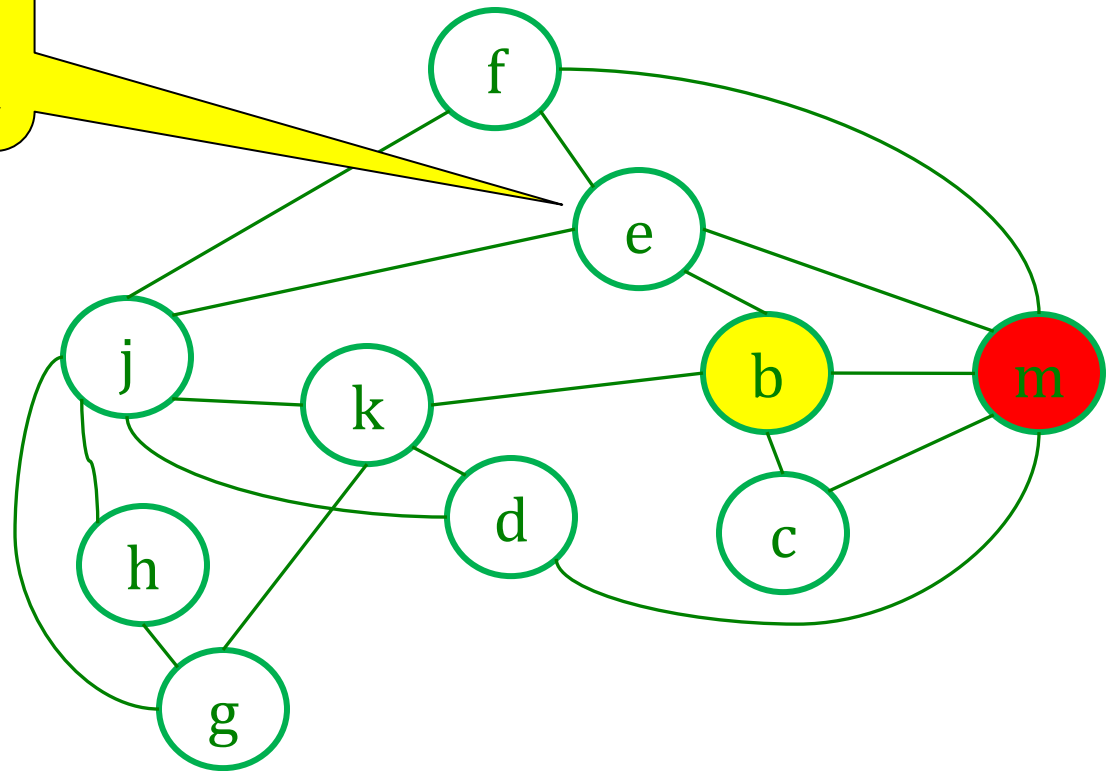
Find a color for this node that's not already used in an adjacent node



Stack: ~~m~~ b e f j d k g h c

# Now, color the nodes in stack order

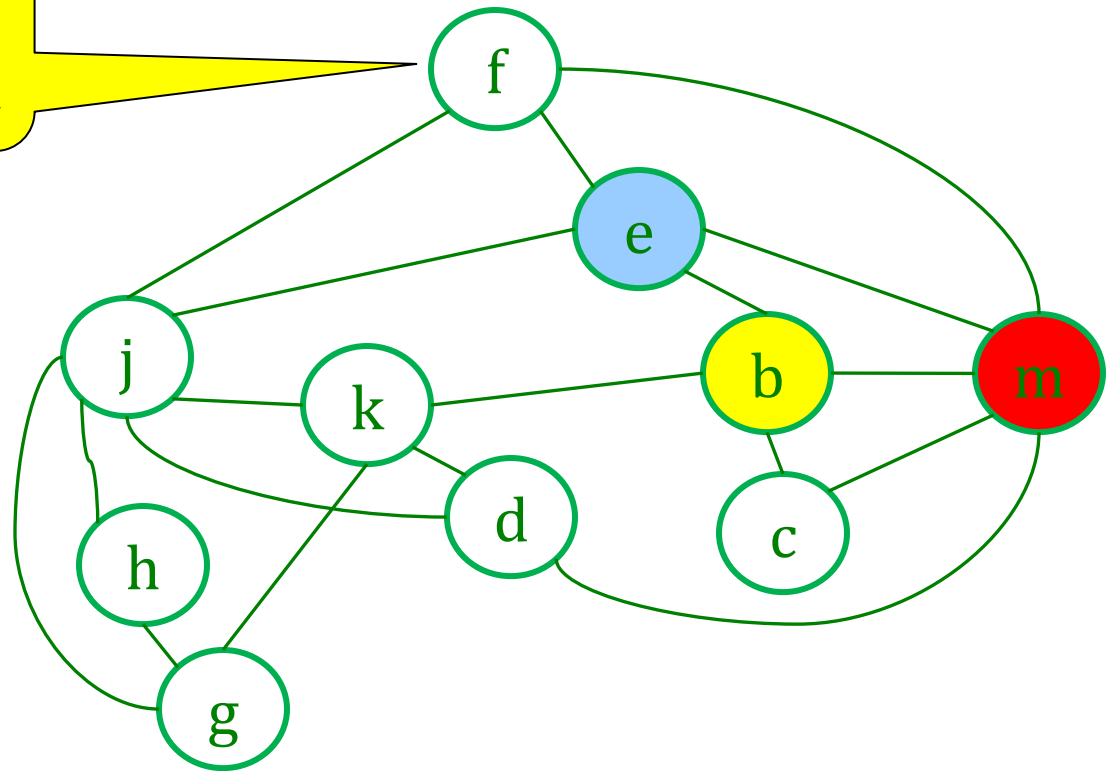
Find a color for this node that's not already used in an adjacent node



Stack: ~~m~~ b e f j d k g h c

# Now, color the nodes in stack order

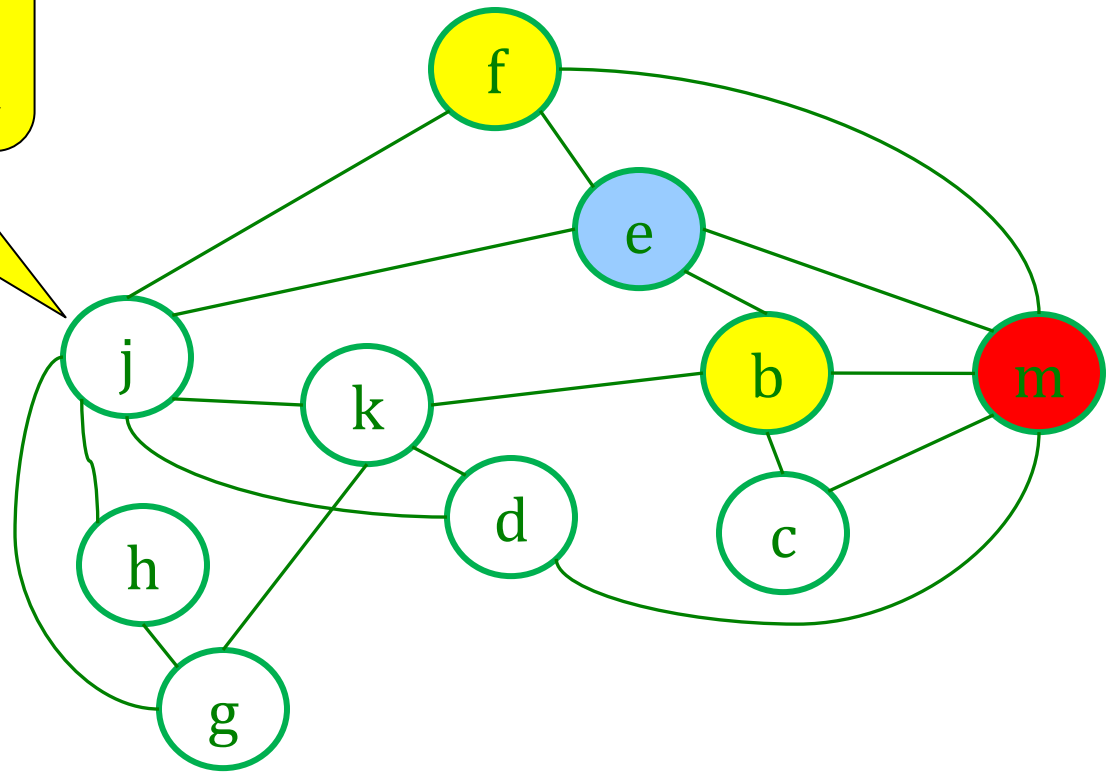
Find a color for this node that's not already used in an adjacent node



Stack: ~~m~~ ~~b~~ e f j d k g h c

# Now, color the nodes in stack order

Find a color for this node that's not already used in an adjacent node

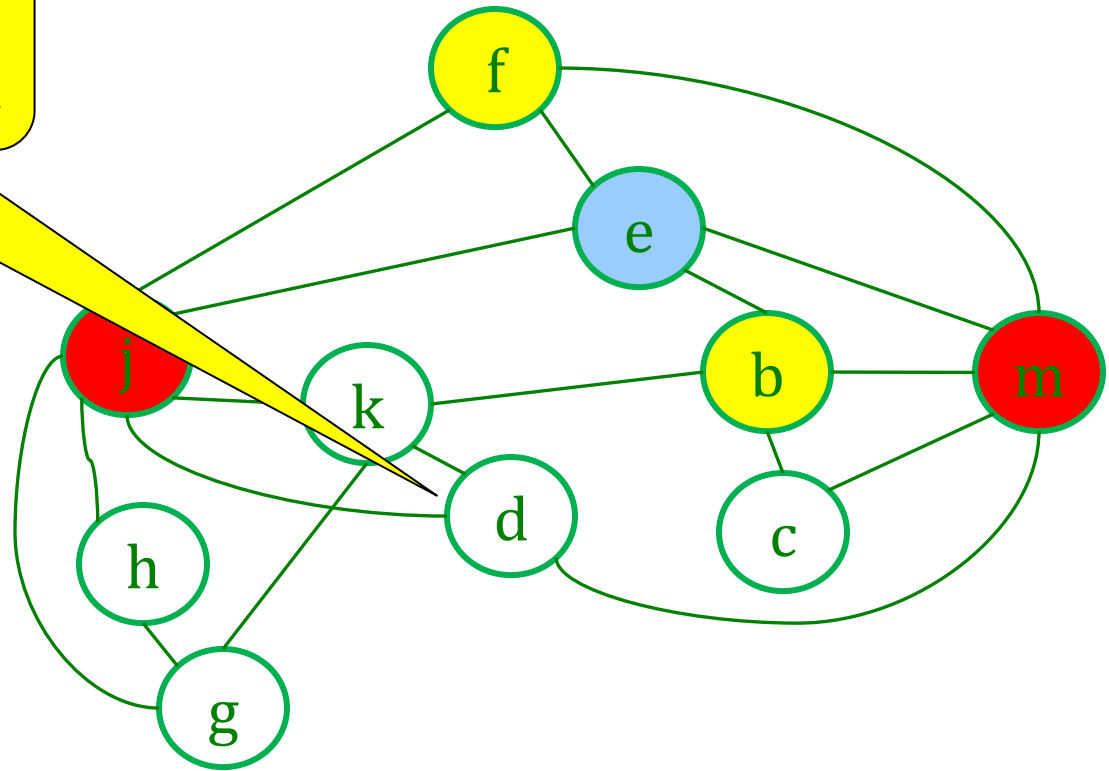


Stack: ~~m~~ ~~b~~ ~~e~~ f j d k g h c



# Now, color the nodes in stack order

Find a color for this node that's not already used in an adjacent node

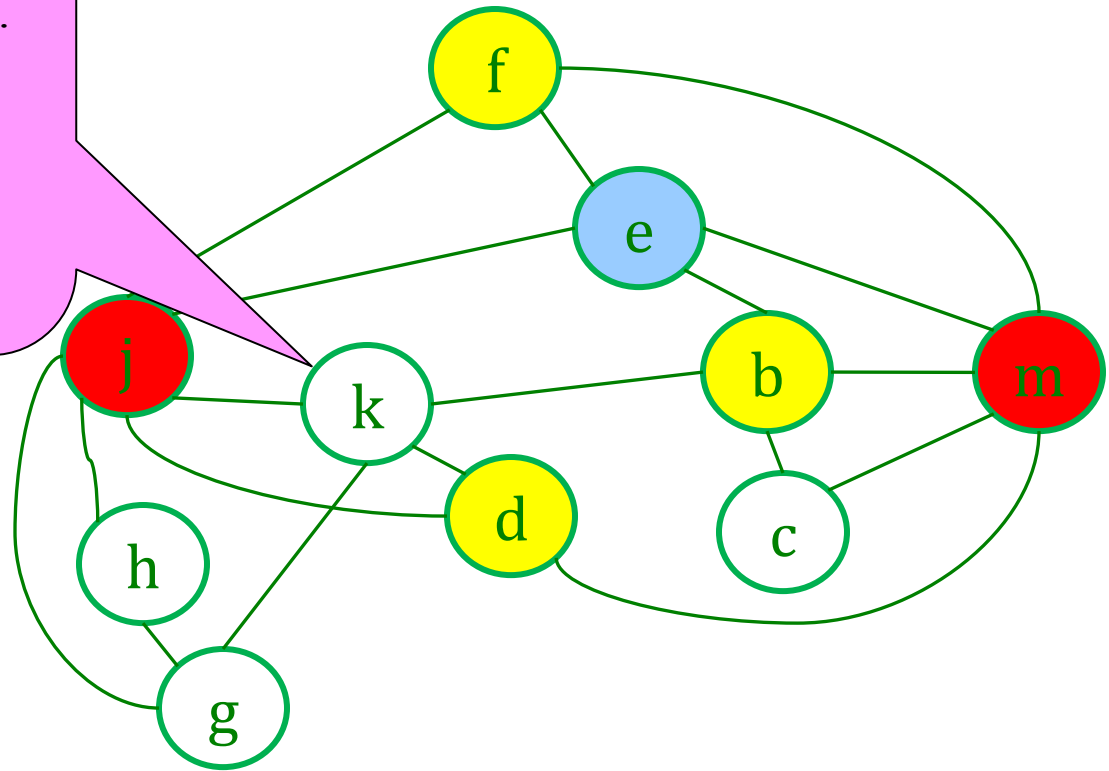


Stack: ~~m~~ ~~b~~ ~~e~~ ~~f~~ d k g h c

# Now, color the nodes in stack order

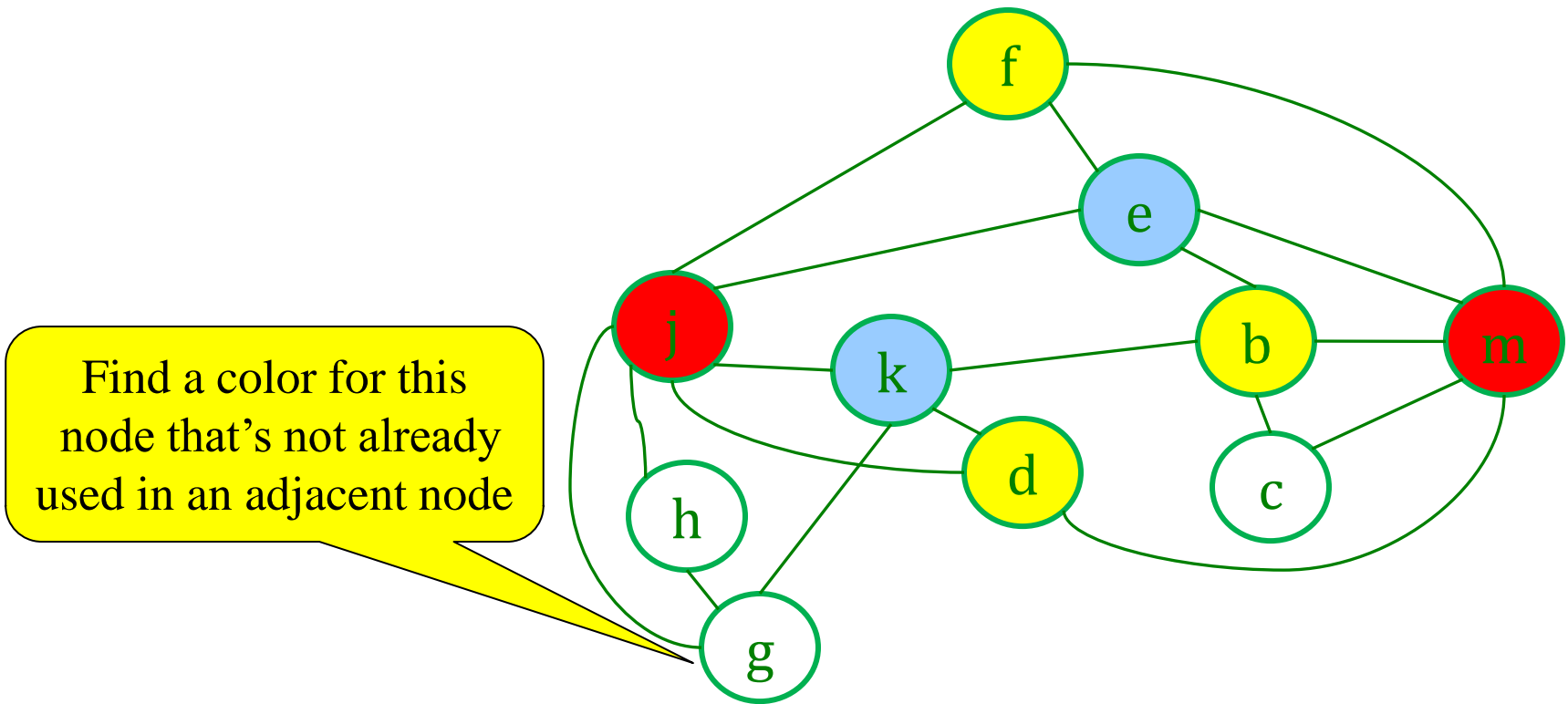
We're about to color node k.  
This was the only one that was  
degree  $\geq 3$  when we removed it.  
Hence, it is not guaranteed that  
we can find a color for it now.

But we got lucky, because  
b and d have the same color!



Stack: ~~m~~ ~~b~~ ~~e~~ ~~f~~ ~~d~~ k g h c

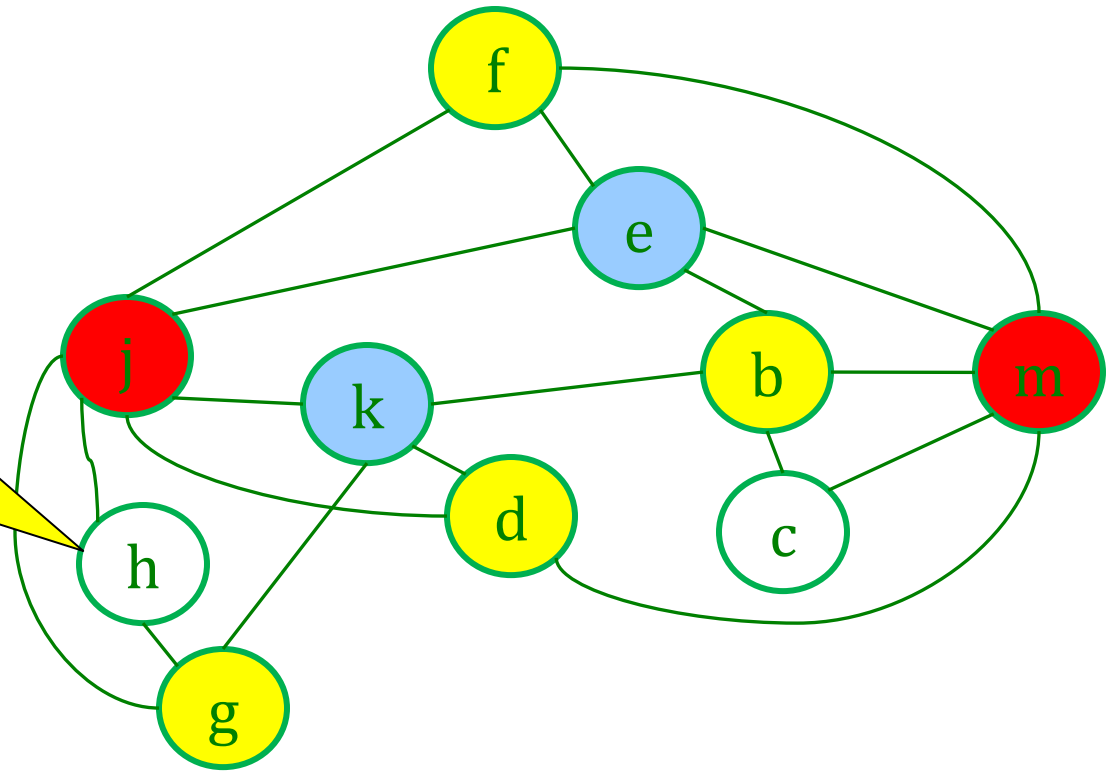
# Now, color the nodes in stack order



Stack: ~~m~~ ~~b~~ ~~e~~ ~~f~~ ~~j~~ ~~d~~ k g h c

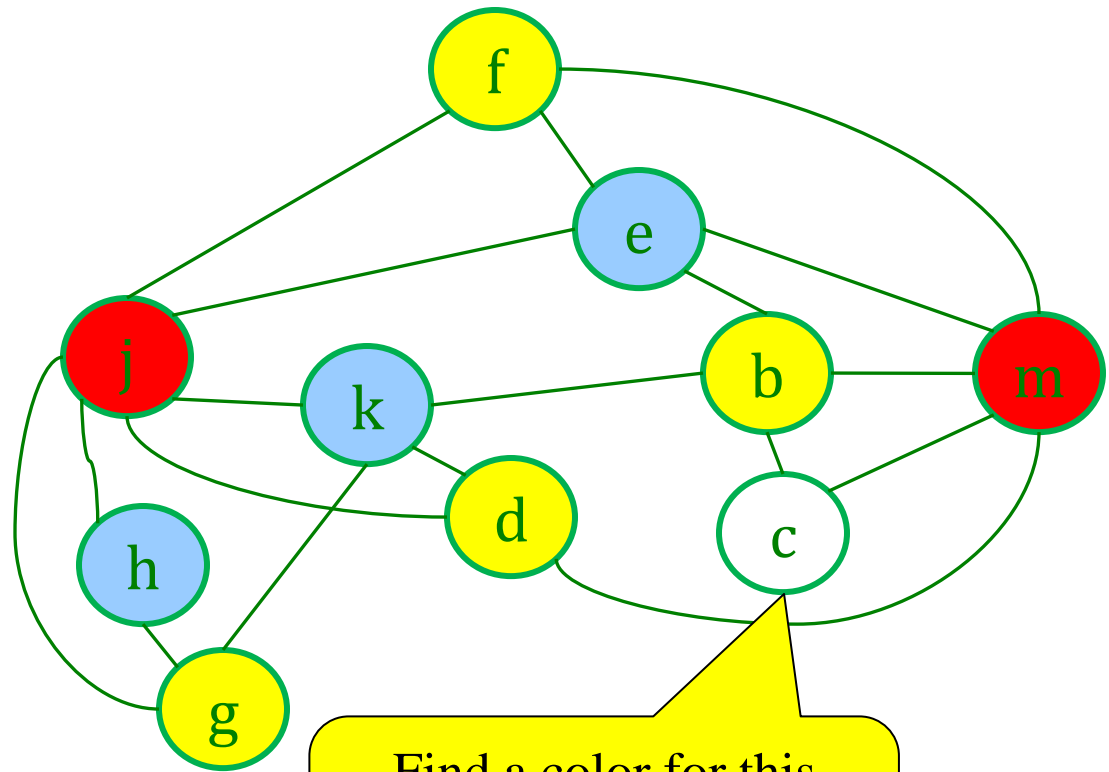
# Now, color the nodes in stack order

Find a color for this node that's not already used in an adjacent node



Stack: ~~m~~ ~~b~~ ~~e~~ ~~f~~ ~~j~~ ~~d~~ ~~k~~ ~~g~~ h c

# Now, color the nodes in stack order



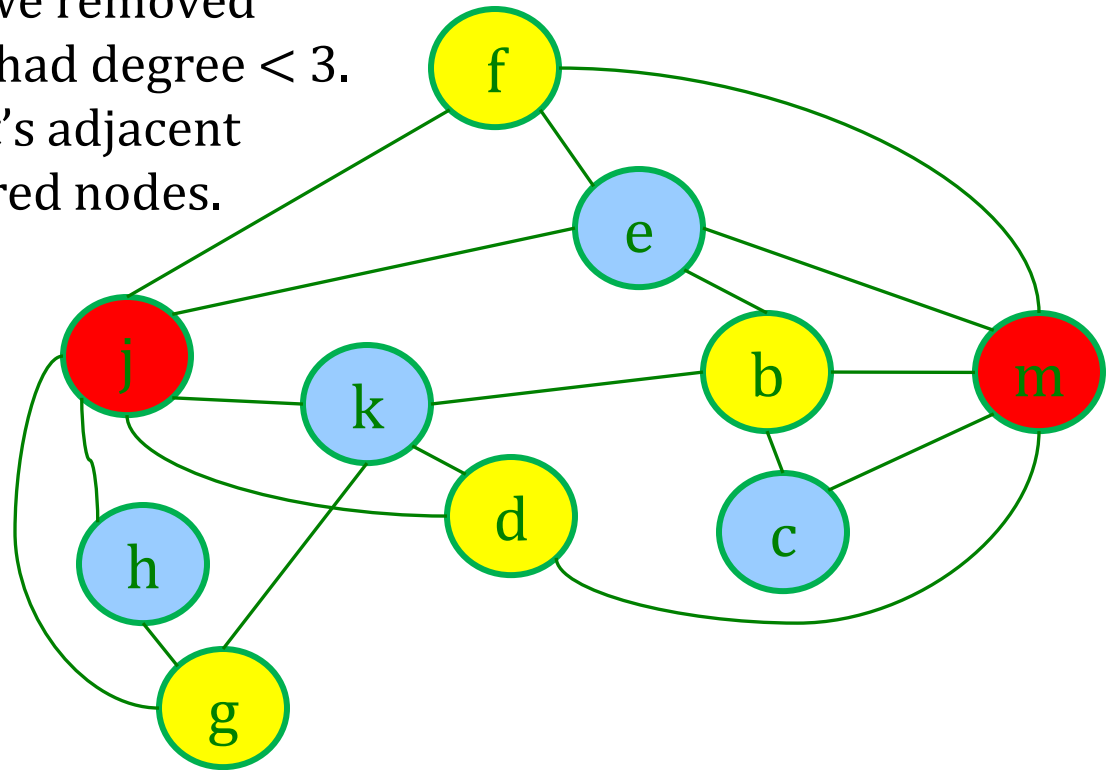
Stack: ~~m b e f j d k g h~~ c

Find a color for this node that's not already used in an adjacent node

# Now, color the nodes in stack order

Why did this work?

Because (usually) when we removed each node, at that time it had degree  $< 3$ .  
So when we put it back, it's adjacent to at most 2 already-colored nodes.

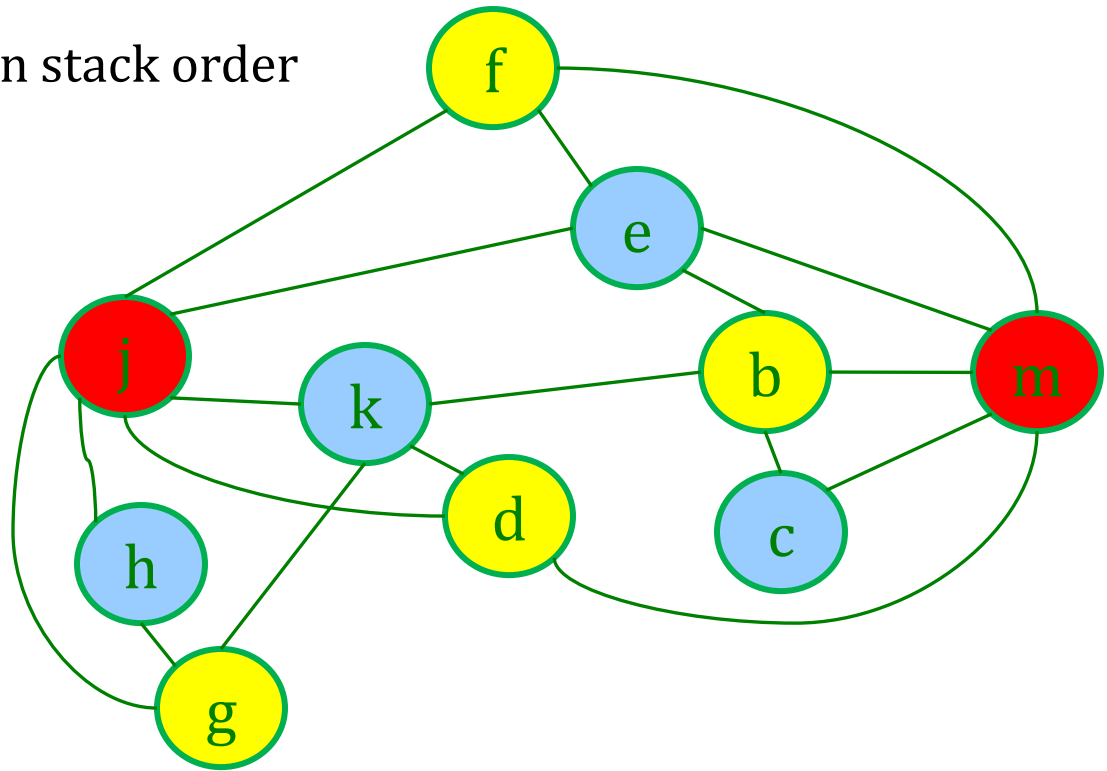


Stack: ~~m~~~~b~~~~e~~~~f~~~~j~~~~d~~~~k~~~~g~~~~h~~~~c~~

# Two-phase algorithm:

Phase 1: list the nodes in some order (“the stack”)

Phase 2: color the nodes in stack order



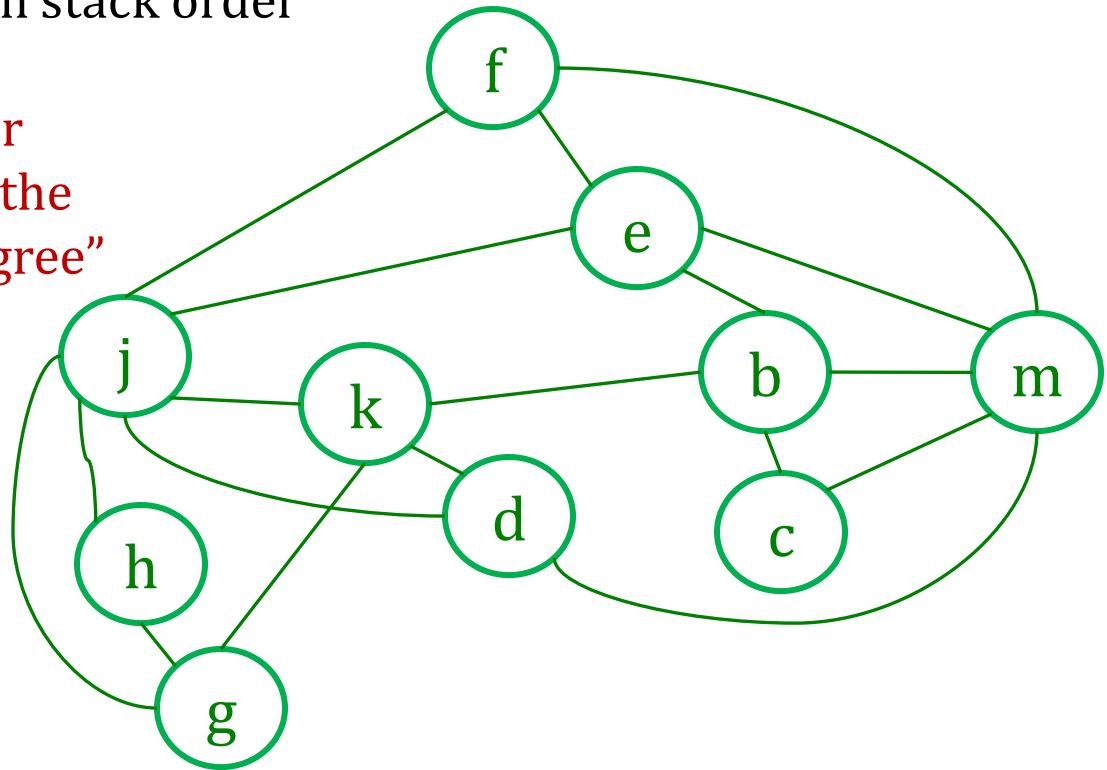
Stack: ~~m~~~~b~~~~e~~~~d~~~~k~~~~j~~~~f~~~~h~~~~g~~~~c~~

# Two phase algorithm

Phase 1: list the nodes in some order (“the stack”)

Phase 2: color the nodes in stack order

What if we use some other coloring order, instead of the “remove nodes of low-degree” order?



Coloring order:



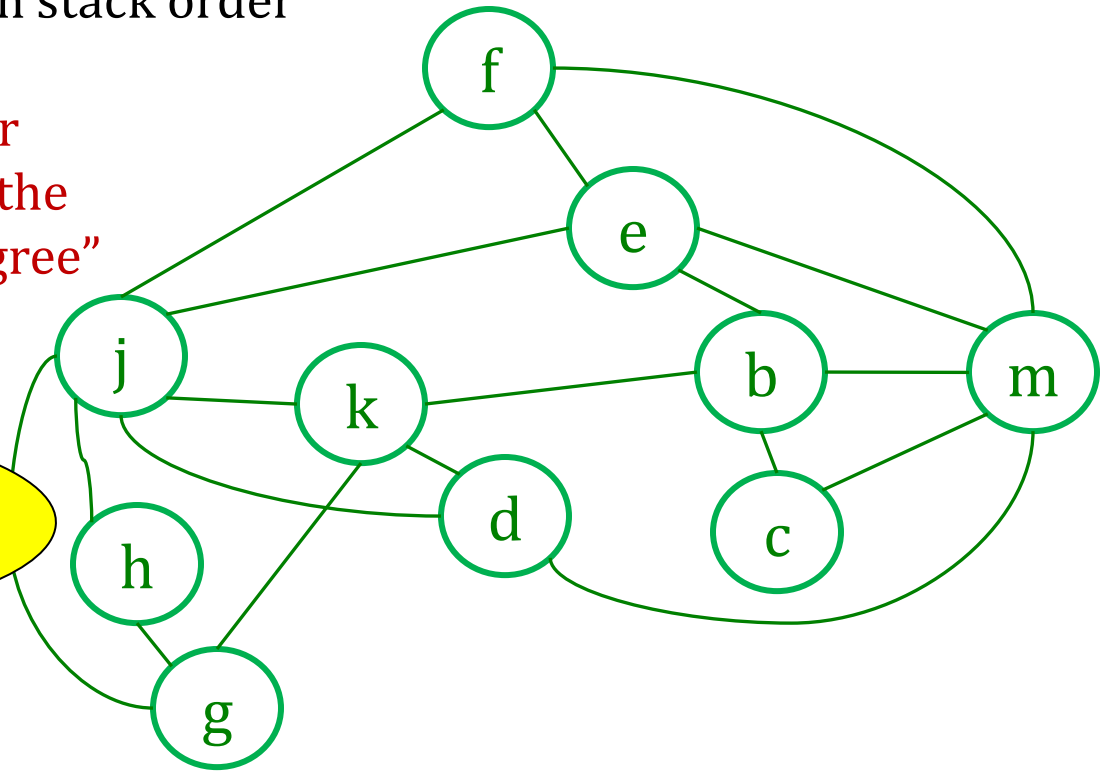
# Two phase algorithm

Phase 1: list the nodes in some order (“the stack”)

Phase 2: color the nodes in stack order

What if we use some other coloring order, instead of the “remove nodes of low-degree” order?

Just for fun, let's use alphabetical order.



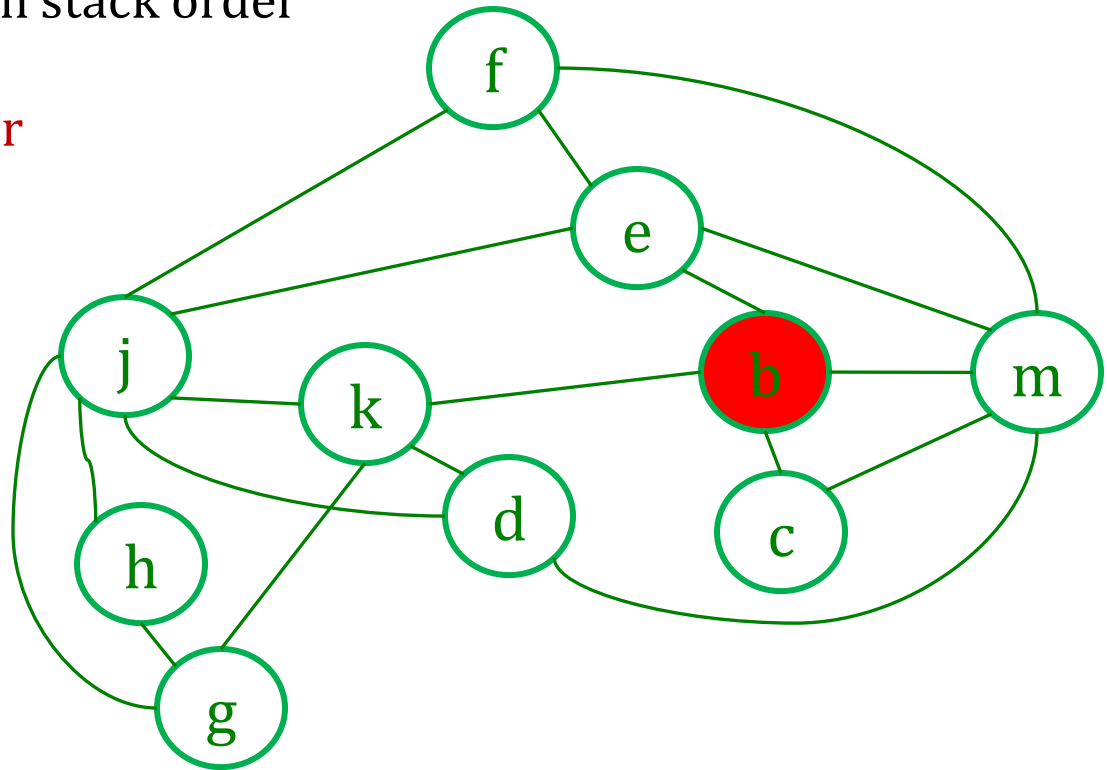
Coloring order: b c d e f g h j k m

# Two phase algorithm

Phase 1: list the nodes in some order (“the stack”)

Phase 2: color the nodes in stack order

What if we use some other coloring order?



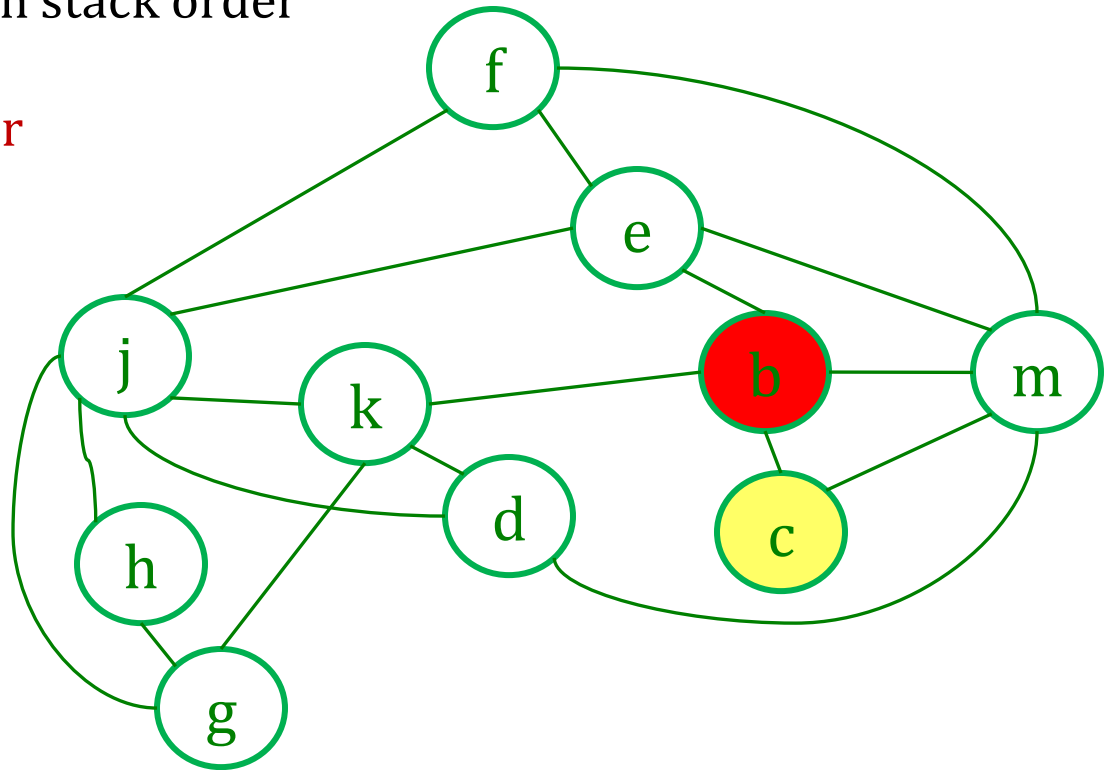
Coloring order: b-c d e f g h j k m

# Two phase algorithm

Phase 1: list the nodes in some order (“the stack”)

Phase 2: color the nodes in stack order

What if we use some other coloring order?



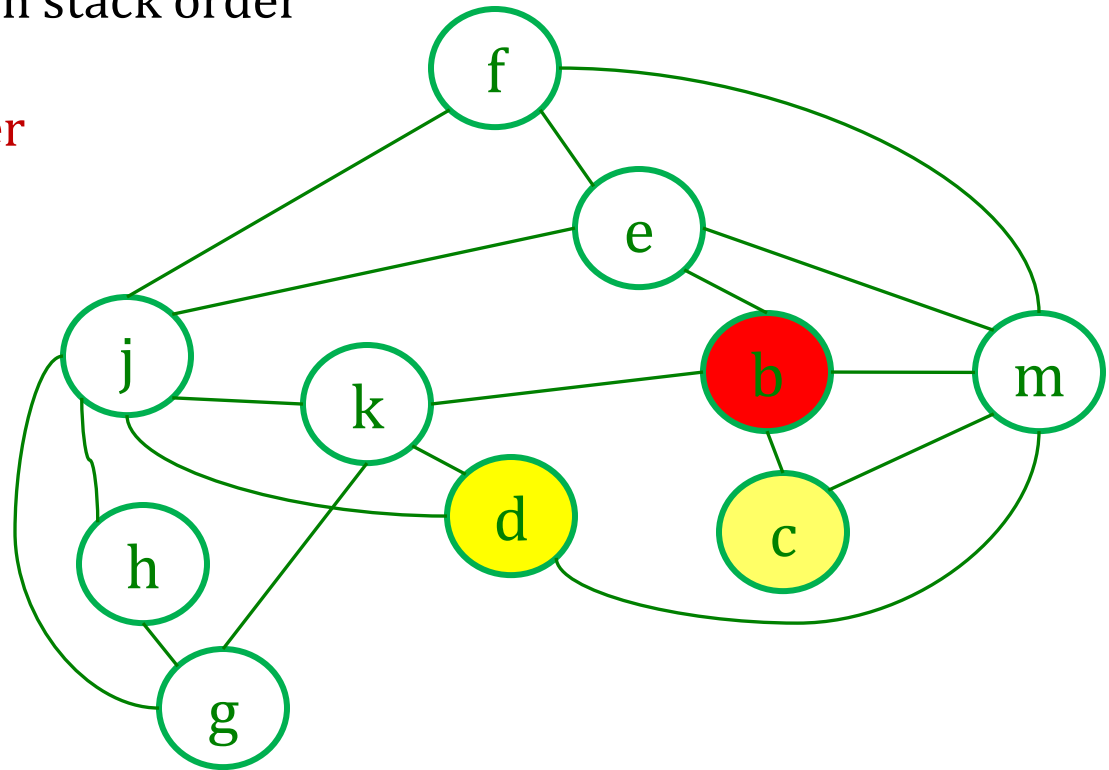
Coloring order: ~~b~~ e d e f g h j k m

# Two phase algorithm

Phase 1: list the nodes in some order (“the stack”)

Phase 2: color the nodes in stack order

What if we use some other coloring order?



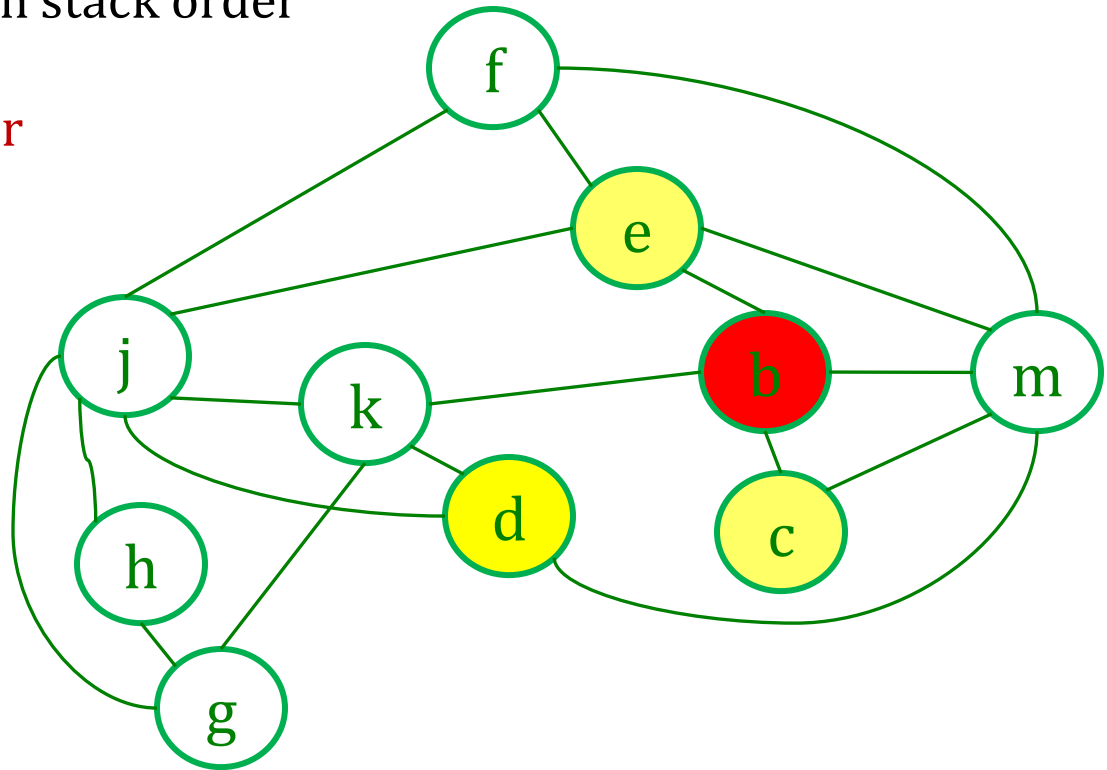
Coloring order: ~~b~~ ~~e~~ ~~d~~ e f g h j k m

# Two phase algorithm

Phase 1: list the nodes in some order (“the stack”)

Phase 2: color the nodes in stack order

What if we use some other coloring order?



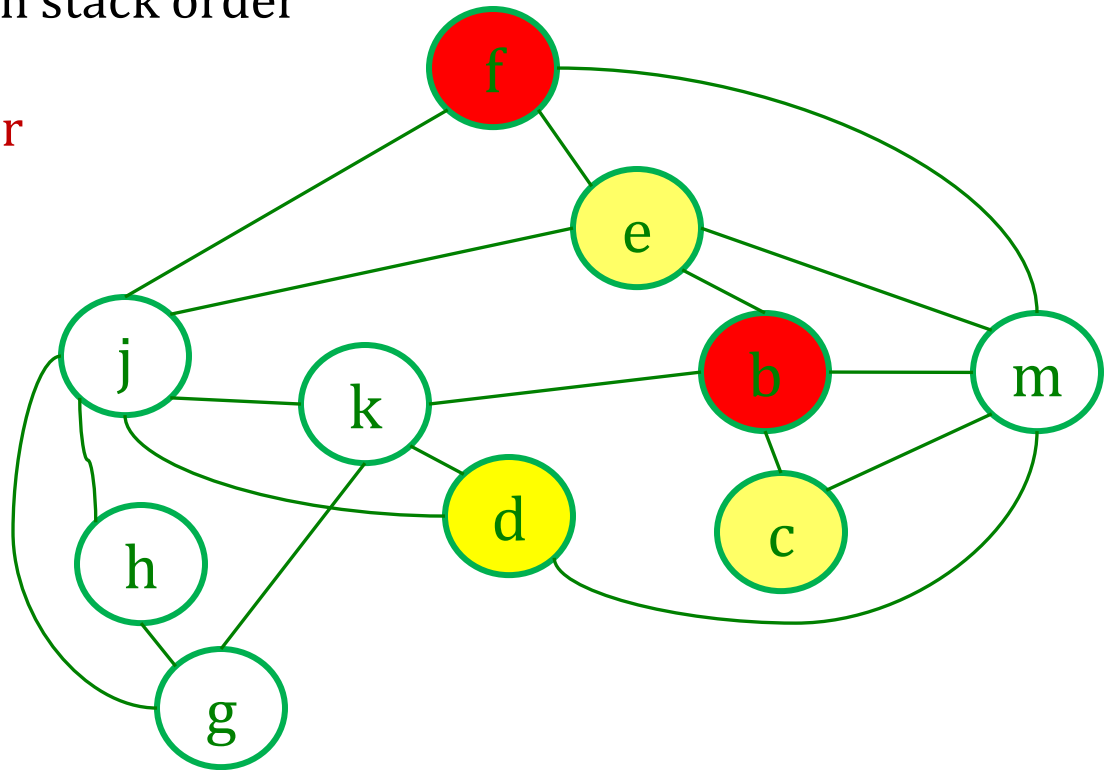
Coloring order: ~~b~~ ~~c~~ ~~d~~ e f g h j k m

# Two phase algorithm

Phase 1: list the nodes in some order (“the stack”)

Phase 2: color the nodes in stack order

What if we use some other coloring order?



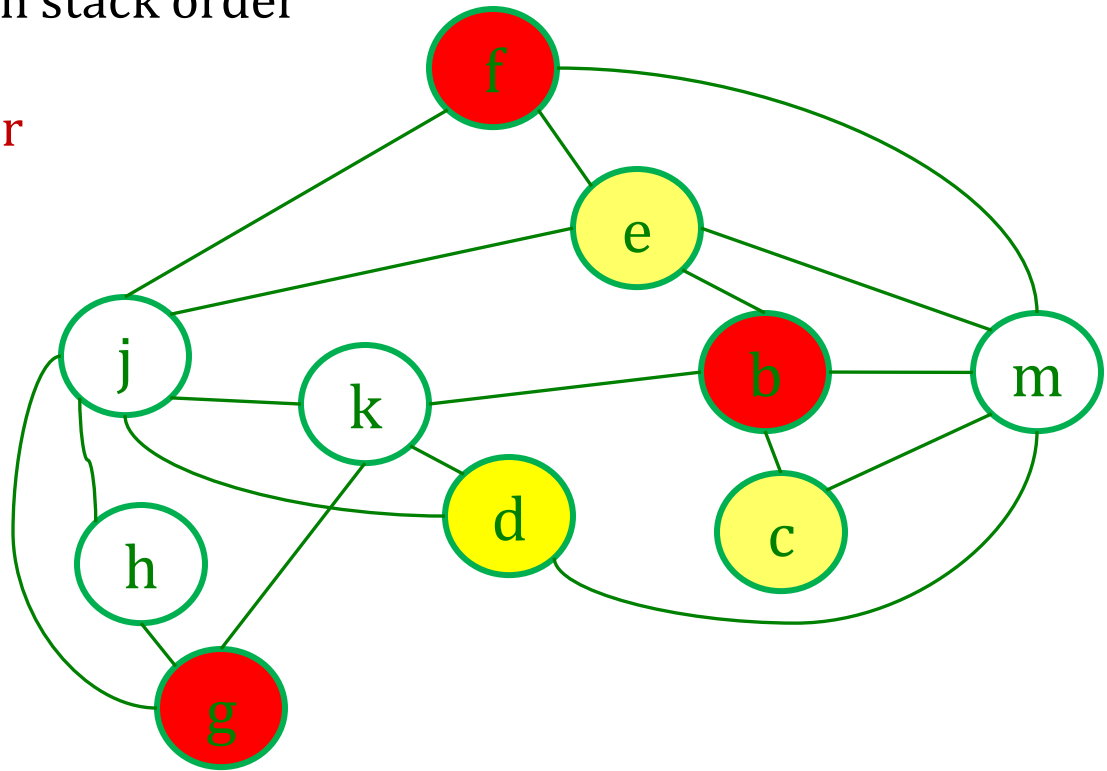
Coloring order: ~~b c d e f~~ g h j k m

# Two phase algorithm

Phase 1: list the nodes in some order (“the stack”)

Phase 2: color the nodes in stack order

What if we use some other coloring order?



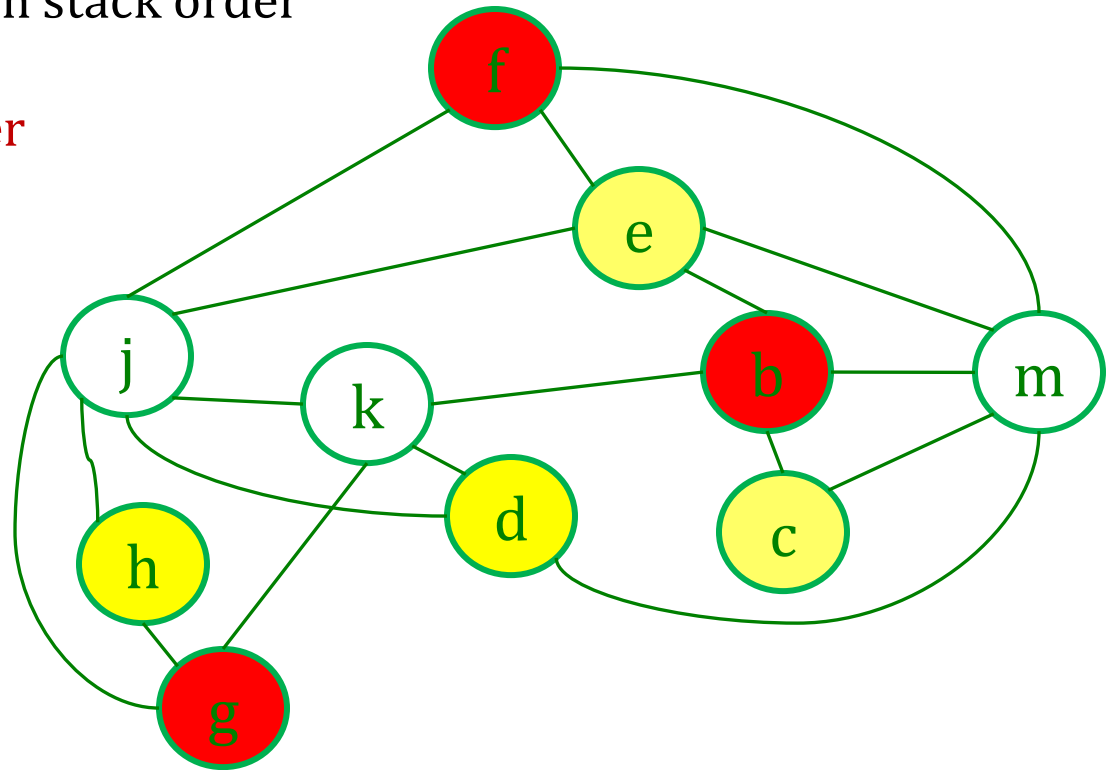
Coloring order: ~~b~~ ~~c~~ ~~d~~ ~~e~~ ~~f~~ ~~g~~ h j k m

# Two phase algorithm

Phase 1: list the nodes in some order (“the stack”)

Phase 2: color the nodes in stack order

What if we use some other coloring order?



Coloring order: ~~b c d e f g h~~ j k m

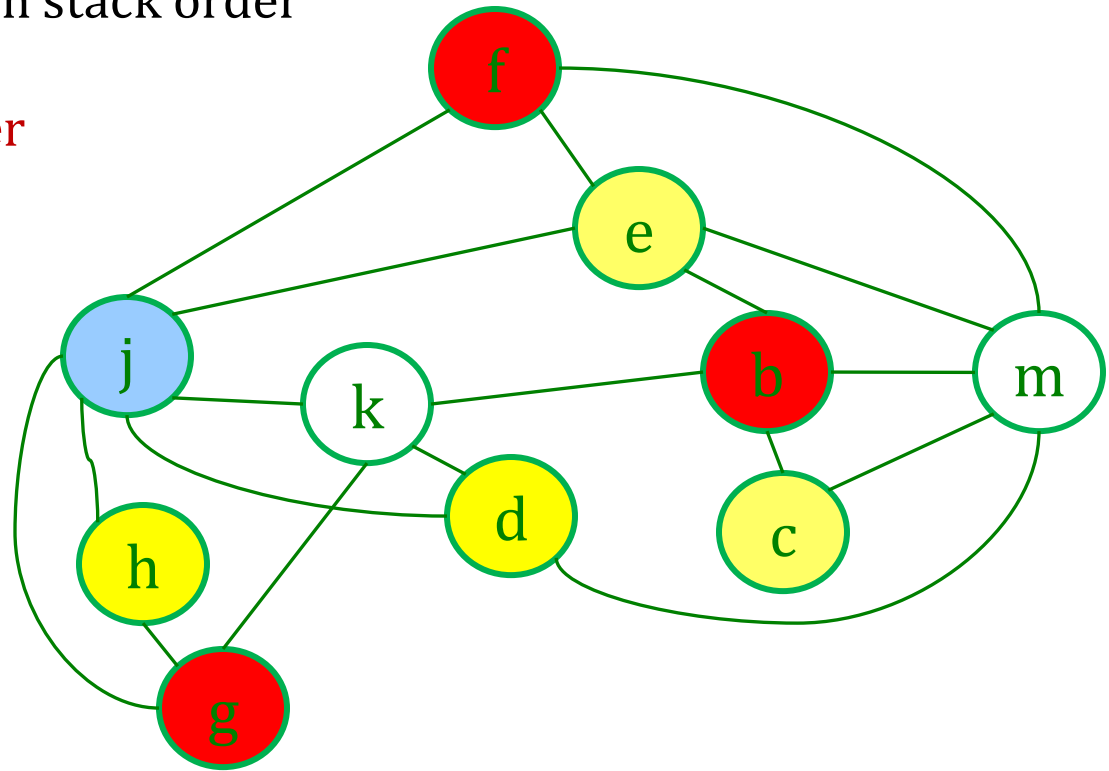


# Two phase algorithm

Phase 1: list the nodes in some order (“the stack”)

Phase 2: color the nodes in stack order

What if we use some other coloring order?



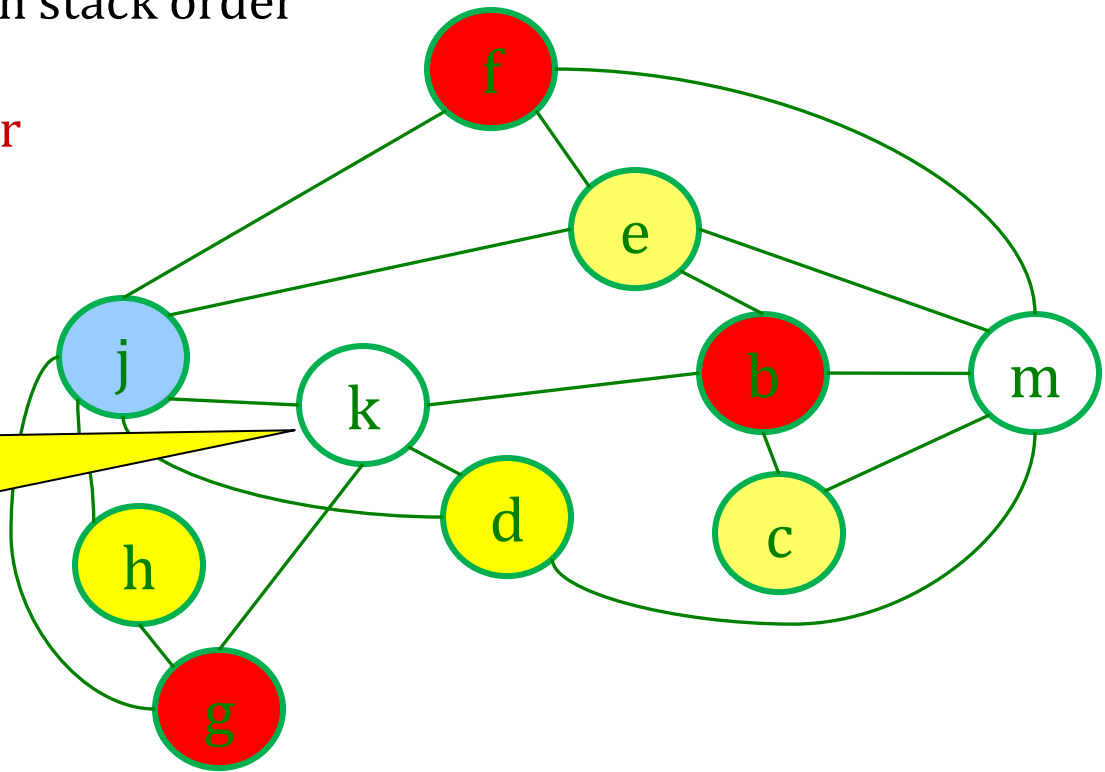
Coloring order: ~~b c d e f g~~ h j k m

# Two phase algorithm

Phase 1: list the nodes in some order (“the stack”)

Phase 2: color the nodes in stack order

What if we use some other coloring order?



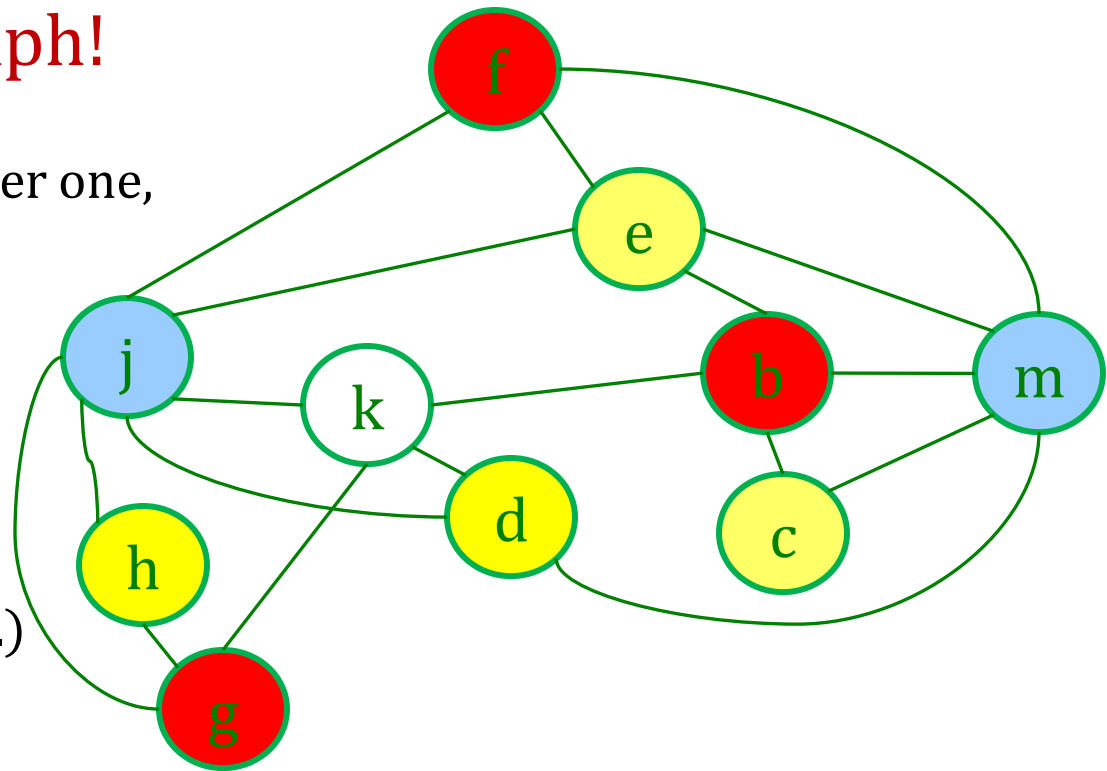
Coloring order: ~~b~~ ~~c~~ ~~d~~ ~~e~~ ~~f~~ ~~g~~ ~~h~~ ~~j~~ k m

# Two phase algorithm

This is a correct *partial* coloring of the graph!

It's not as good as the other one, but it is correct.

(In a register-allocation application, variable *k* would not be assigned a register, but would be spilled to the stack frame.)

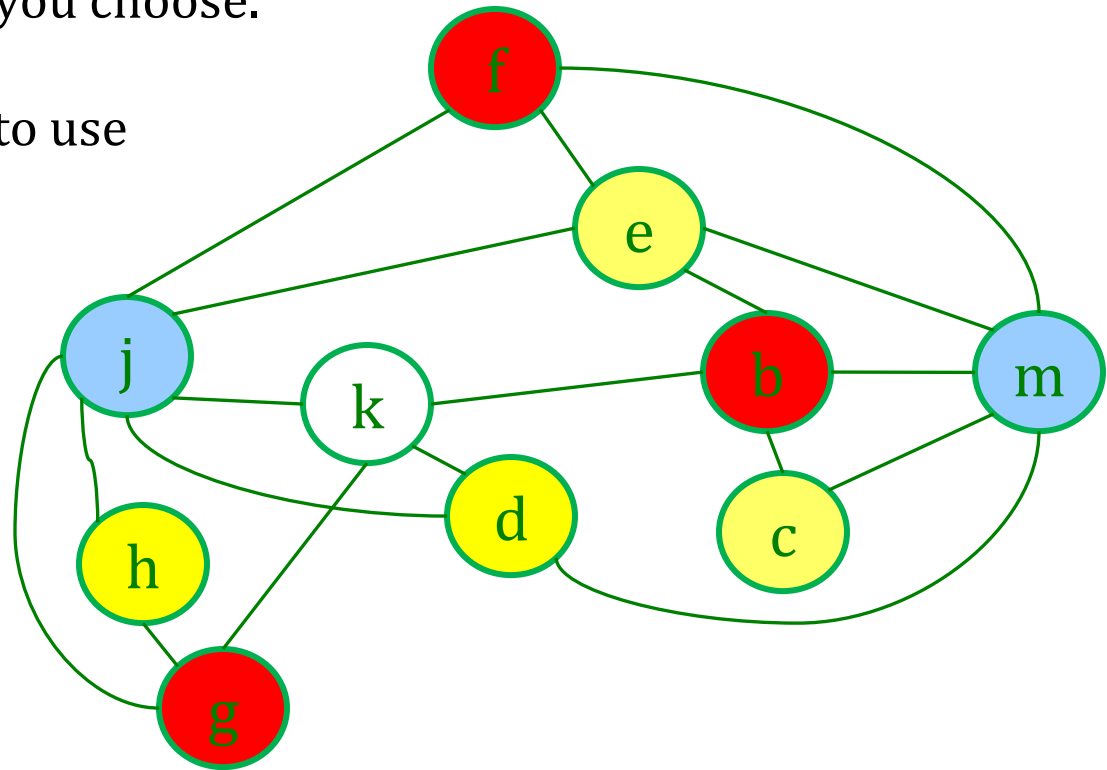


Coloring order: ~~b~~ ~~c~~ ~~d~~ ~~e~~ ~~f~~ ~~g~~ ~~h~~ ~~j~~ ~~k~~ ~~m~~

# Two phase algorithm

Moral: The two-phase algorithm is correct no matter what ordering you choose.

In phase 1, not necessary to use Kempe's algorithm, although that may give better results.



Coloring order: ~~b~~ ~~c~~ ~~d~~ ~~e~~ ~~f~~ ~~g~~ ~~h~~ ~~j~~ ~~k~~ ~~m~~

# Implications for program verification

Moral: The two-phase algorithm is correct no matter what ordering you choose.

In phase 1, not necessary to use Kempe's algorithm, although that may give better results.

**Therefore:** When proving the correctness of this graph-coloring algorithm, we do not have to prove that the ordering phase correctly follows Kempe's algorithm; any ordering will do! We just have to prove things about phase 2.\*

\* This was once pointed out to me by an anonymous referee named G.G.

# Representing graphs in a functional program

Node labels: b,c,d,e,...

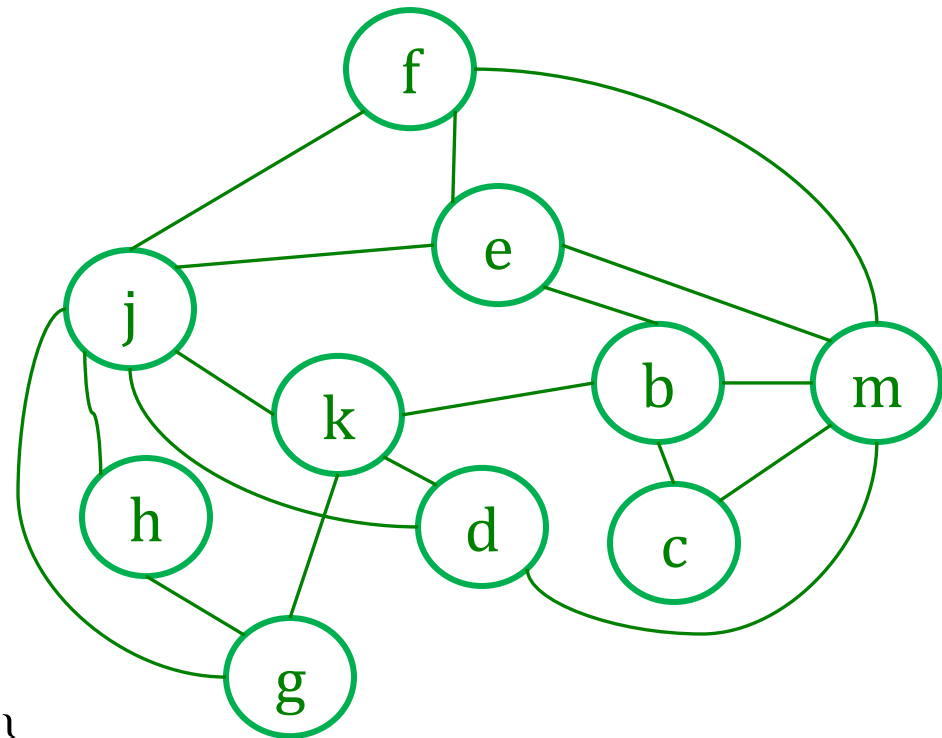
Edges of node f: {j,e,m}

This is a  
*set of nodes*

Graph is a *finite function*

from node to set-of-nodes:

[  $b \mapsto \{c,e,k,m\}$ ,  $c \mapsto \{b,m\}$ ,  $d \mapsto \{j,k,m\}$ ,  
 $e \mapsto \{b,f,j,m\}$ ,  $f \mapsto \{e,j,m\}$ ,  $g \mapsto \{h,j,k\}$ ,  
 $h \mapsto \{g,j\}$ ,  $j \mapsto \{d,e,f,g,h,k\}$ ,  $k \mapsto \{b,d,g,j\}$ ,  
 $m \mapsto \{b,c,d,e,f\}$  ]



# Undirected graphs

Graph coloring is done on *undirected* graphs.

In an undirected graph, whenever  $x \rightarrow y$  then  $y \rightarrow x$ .

Or we can write,

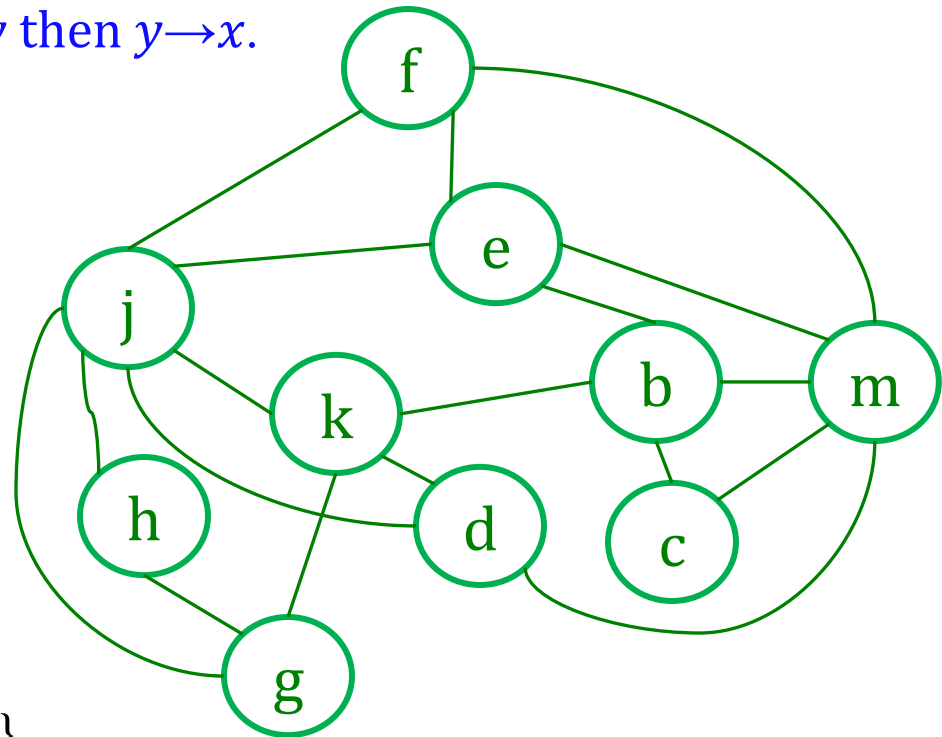
$\text{undirected}(G) :=$

$$\forall x, y. y \in G(x) \Rightarrow x \in G(y)$$

Graph is a *finite function*

from node to set-of-nodes:

$$\begin{aligned} & [ b \mapsto \{c, e, k, m\}, c \mapsto \{b, m\}, d \mapsto \{j, k, m\}, \\ & e \mapsto \{b, f, j, m\}, f \mapsto \{e, j, m\}, g \mapsto \{h, j, k\}, \\ & h \mapsto \{g, j\}, j \mapsto \{d, e, f, g, h, k\}, k \mapsto \{b, d, g, j\}, \\ & m \mapsto \{b, c, d, e, f\} ] \end{aligned}$$



# Sets and maps

We can use Coq's FSets and FMaps libraries to implement (efficient) functional sets and functional maps over small-integer keys:

Module E := PositiveOrderedTypeBits. (*\* E for "Element type," positive numbers \**)

Module S := PositiveSet. (*\* finite sets of positive numbers \**)

Module M := PositiveMap. (*\* finite functions from positive numbers to arbitrary type \**)

Graph is a *finite function* from node to set-of-nodes:

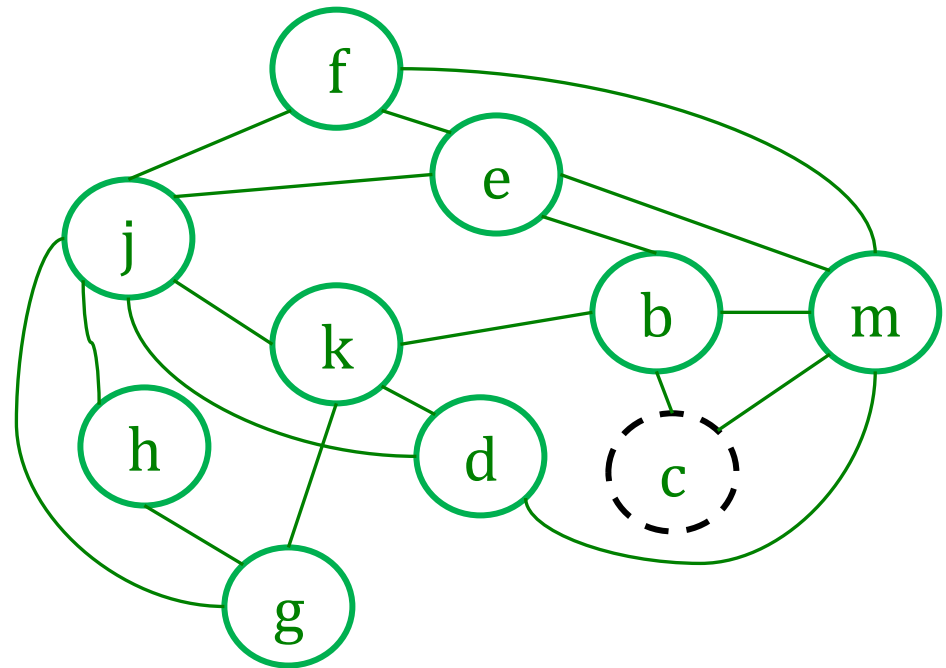
[  $b \mapsto \{c, e, k, m\}$ ,  $c \mapsto \{b, m\}$ ,  $d \mapsto \{j, k, m\}$ , ... ]



# Removing a node from a graph

Definition `remove_node (c: node) (G: graph) : graph :=  
M.map (S.remove c) (M.remove c G).`

First, remove `c`  
from finite-map `G`



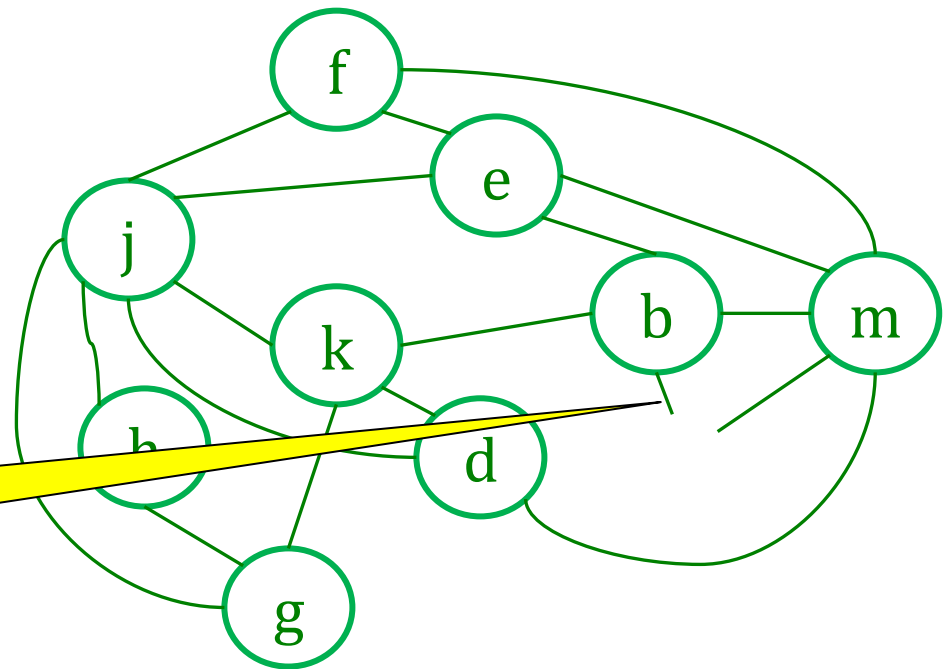
[  $b \mapsto \{c, e, k, m\}$ ,  $c \mapsto \{b, m\}$ ,  $d \mapsto \{j, k, m\}$ ,  
 $e \mapsto \{b, f, j, m\}$ ,  $f \mapsto \{e, j, m\}$ ,  $g \mapsto \{h, j, k\}$ ,  
 $h \mapsto \{g, j\}$ ,  $j \mapsto \{d, e, f, g, h, k\}$ ,  $k \mapsto \{b, d, g, j\}$ , 57  
 $m \mapsto \{b, c, d, e, f\}$  ]

# Removing a node from a graph

Definition `remove_node (c: node) (G: graph) : graph :=  
M.map (S.remove c) (M.remove c G).`

First, remove `c`  
from finite-map `G`

This leaves some  
dangling edges



[  $b \mapsto \{c, e, k, m\}$ ,  $c \mapsto \{b, m\}$ ,  $d \mapsto \{j, k, m\}$ ,  
 $e \mapsto \{b, f, j, m\}$ ,  $f \mapsto \{e, j, m\}$ ,  $g \mapsto \{h, j, k\}$ ,  
 $h \mapsto \{g, j\}$ ,  $j \mapsto \{d, e, f, g, h, k\}$ ,  $k \mapsto \{b, d, g, j\}$ ,  
 $m \mapsto \{b, c, d, e, f\}$  ] 58

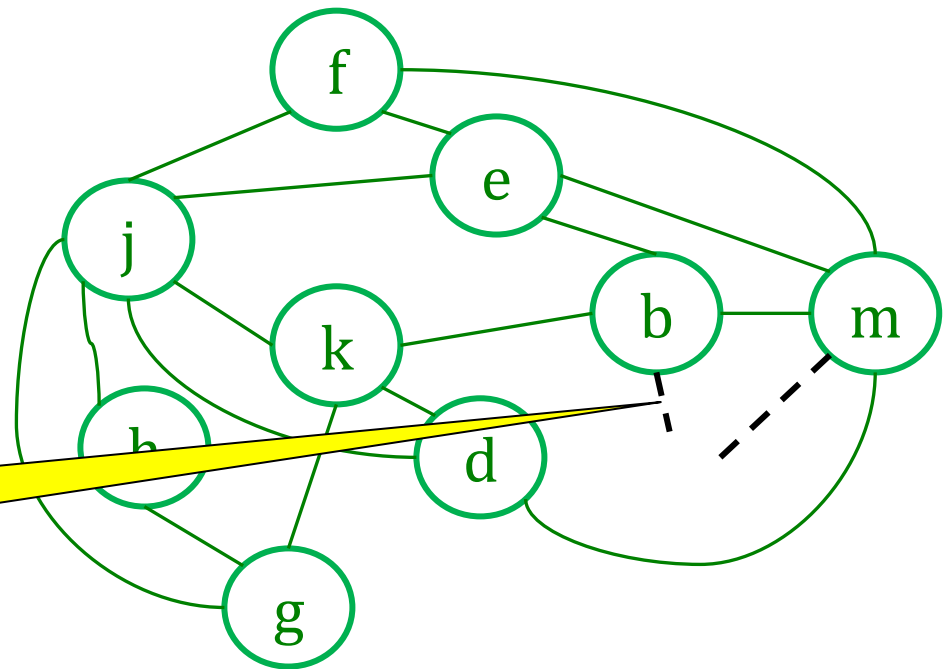
# Removing a node from a graph

Definition `remove_node (c: node) (G: graph) : graph :=`  
`M.map (S.remove c) (M.remove c G).`

First, remove `c`  
from finite-map `G`

This leaves some  
dangling edges

remove the  
dangling edges



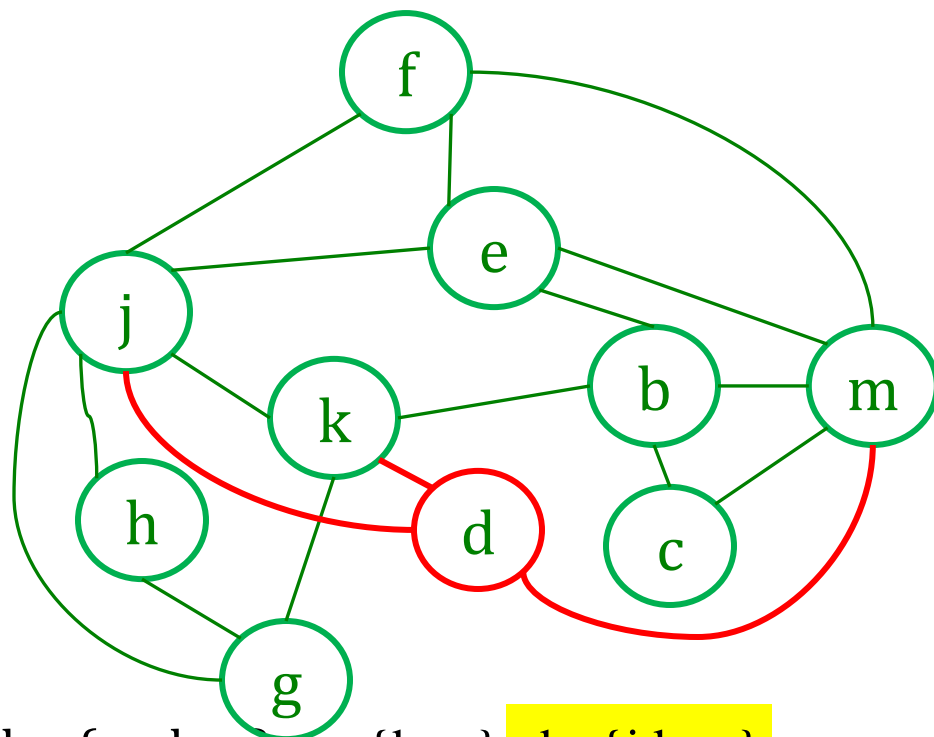
[  $b \mapsto \{c, e, k, m\}$ ,  $c \mapsto \{b, m\}$ ,  $d \mapsto \{j, k, m\}$ ,  
 $e \mapsto \{b, f, j, m\}$ ,  $f \mapsto \{e, j, m\}$ ,  $g \mapsto \{h, j, k\}$ ,  
 $h \mapsto \{g, j\}$ ,  $j \mapsto \{d, e, f, g, h, k\}$ ,  $k \mapsto \{b, d, g, j\}$ ,  
 $m \mapsto \{b, c, d, e, f\}$  ]

# Testing whether a node is low-degree

Definition  $\text{low\_deg} (K: \text{nat}) (n: \text{node}) (\text{adj}: \text{nodeset}) : \text{bool} :=$   
 $\text{S.cardinal adj} <? K.$

Example:  $K=3, n=d, \text{adj}=\{j,k,m\}$

$(\text{S.cardinal } \{j,k,m\} <? 3)$  is **false**.



$[ b \mapsto \{c,e,k,m\}, c \mapsto \{b,m\}, d \mapsto \{j,k,m\},$   
 $e \mapsto \{b,f,j,m\}, f \mapsto \{e,j,m\}, g \mapsto \{h,j,k\},$   
 $h \mapsto \{g,j\}, j \mapsto \{d,e,f,g,h,k\}, k \mapsto \{b,d,g,j\},$  60  
 $m \mapsto \{b,c,d,e,f\} ]$

# The fold function on a finite map

M.fold:  $\forall A B : \text{Type}, (\text{M.el}t \rightarrow A \rightarrow B \rightarrow B) \rightarrow \text{M.t } A \rightarrow B \rightarrow B$

M.fold:  $(\text{node} \rightarrow \text{nodeset} \rightarrow \text{nodeset} \rightarrow \text{nodeset}) \rightarrow \text{graph} \rightarrow \text{nodeset} \rightarrow \text{nodeset}$

*(\* calculate the set of those nodes of G that satisfy predicate P \*)*

Definition subset\_nodes (P: node  $\rightarrow$  nodeset  $\rightarrow$  bool) (g: graph) :=

M.fold (fun n adj s => if P n adj then S.add n s else s) g S.empty.

[ b $\mapsto$ {c,e,k,m}, c $\mapsto$ {b,m}, d $\mapsto$ {j,k,m},  
e $\mapsto$ {b,f,j,m}, f $\mapsto$ {e,j,m}, g $\mapsto$ {h,j,k},  
h $\mapsto$ {g,j}, j $\mapsto$ {d,e,f,g,h,k}, k $\mapsto$ {b,d,g,j}, 61  
m $\mapsto$ {b,c,d,e,f} ]

# The set of low-degree nodes

M.fold:  $\forall A B : \text{Type}, (\text{M.elt} \rightarrow A \rightarrow B \rightarrow B) \rightarrow \text{M.t } A \rightarrow B \rightarrow B$

M.fold:  $(\text{node} \rightarrow \text{nodeset} \rightarrow \text{nodeset} \rightarrow \text{nodeset}) \rightarrow \text{graph} \rightarrow \text{nodeset} \rightarrow \text{nodeset}$

*(\* calculate the set of those nodes of G that satisfy predicate P \*)*

Definition subset\_nodes (P: node  $\rightarrow$  nodeset  $\rightarrow$  bool) (g: graph) :=

M.fold (fun n adj s => if P n adj then S.add n s else s) g S.empty.

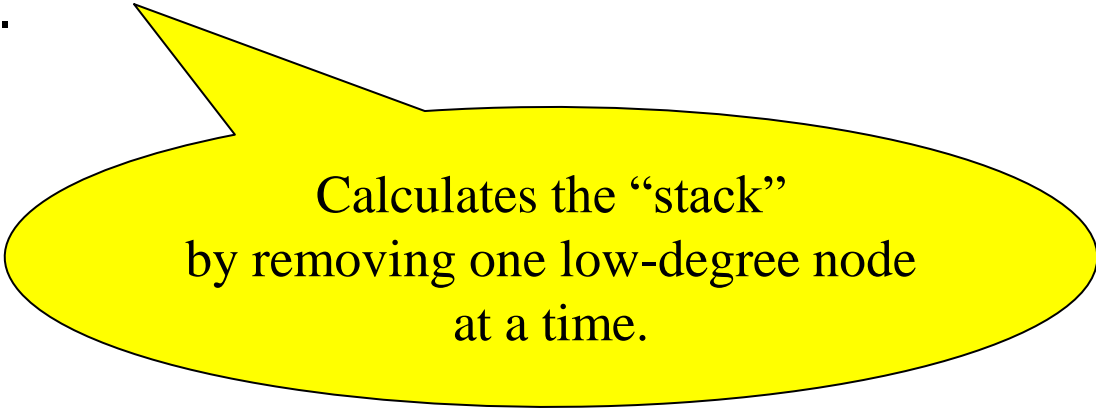
(subset\_nodes (low\_deg K) G)

The set of low-degree  
nodes of G

[ b $\mapsto$ {c,e,k,m}, c $\mapsto$ {b,m}, d $\mapsto$ {j,k,m},  
e $\mapsto$ {b,f,j,m}, f $\mapsto$ {e,j,m}, g $\mapsto$ {h,j,k},  
h $\mapsto$ {g,j}, j $\mapsto$ {d,e,f,g,h,k}, k $\mapsto$ {b,d,g,j},  
m $\mapsto$ {b,c,d,e,f} ]

# Phase 1 of Kempe's algorithm

```
Function select (K: nat) (G: graph) : list node :=  
  match S.choose (subset_nodes (low_deg K) G) with  
  | Some n ⇒ n :: select K (remove_node n G)  
  | None ⇒ nil  
end.
```



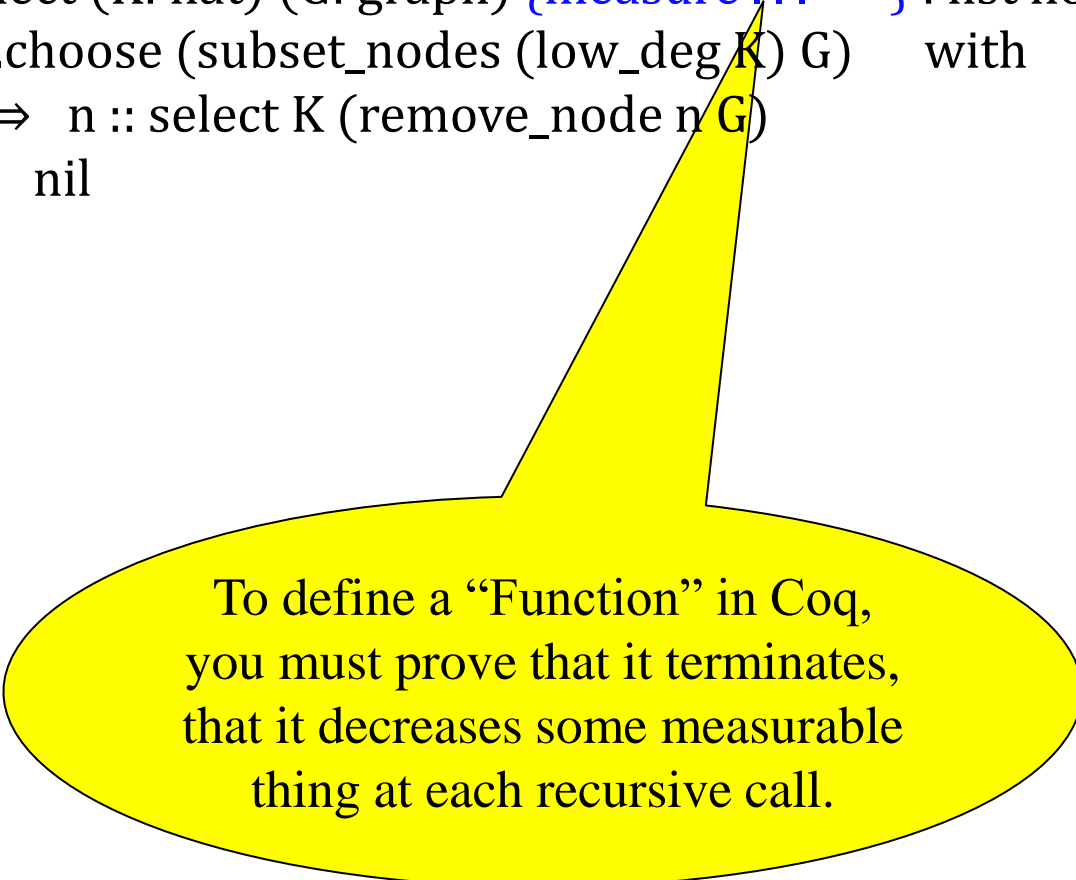
Calculates the “stack”  
by removing one low-degree node  
at a time.

# Recursive functions in Coq

```
Function select (K: nat) (G: graph) {measure ... } : list node :=  
  match S.choose (subset_nodes (low_deg K) G) with  
  | Some n ⇒ n :: select K (remove_node n G)  
  | None ⇒ nil  
end.
```

Proof. ...

Defined.



To define a “Function” in Coq,  
you must prove that it terminates,  
that it decreases some measurable  
thing at each recursive call.



# Recursive functions in Coq

```
Function select (K: nat) (G: graph) {measure M.cardinal G} : list node :=  
  match S.choose (subset_nodes (low_deg K) G) with  
  | Some n ⇒ n :: select K (remove_node n G)  
  | None ⇒ nil  
  end.
```

Proof. apply select\_terminates.

Defined.

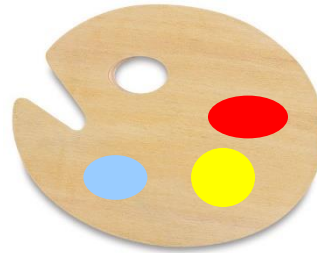
Lemma select\_terminates:

$$\forall (K: \text{nat}) (G : \text{graph}) (n : \text{node}),$$
$$\text{S.choose (subset\_nodes (low\_deg K) G) = Some } n \rightarrow$$
$$\text{M.cardinal (remove\_node } n \text{ G) } < \text{M.cardinal G.}$$

# Color palette



For 6-coloring a graph  
(or painting a portrait)

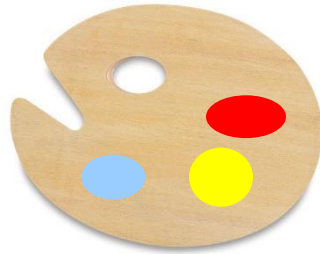


For 3-coloring a graph  
(not so good for a portrait)

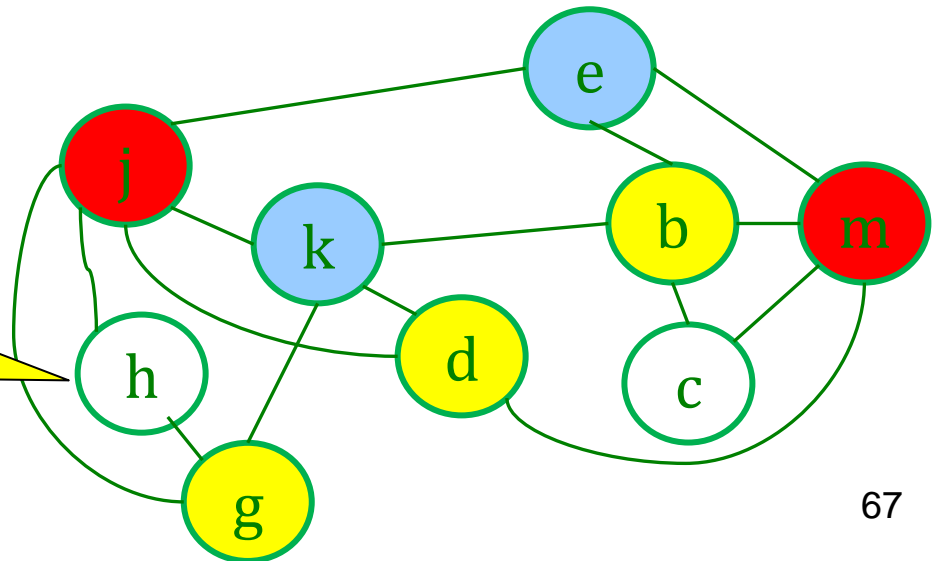
palette: S.t

A “palette” is a *set* of colors

# Phase 2 of the algorithm



Find a color for this node that's not already used in an adjacent node



# Phase 2 of the algorithm

Definition coloring := M.t node.

Definition colors\_of (f: coloring) (s: S.t) : S.t :=

S.fold

(fun n s => match M.find n f with Some c => S.add c s | None => s end)

s S.empty.

Definition color1 (palette: S.t) (g: graph) (n: node) (f: coloring) : coloring :=

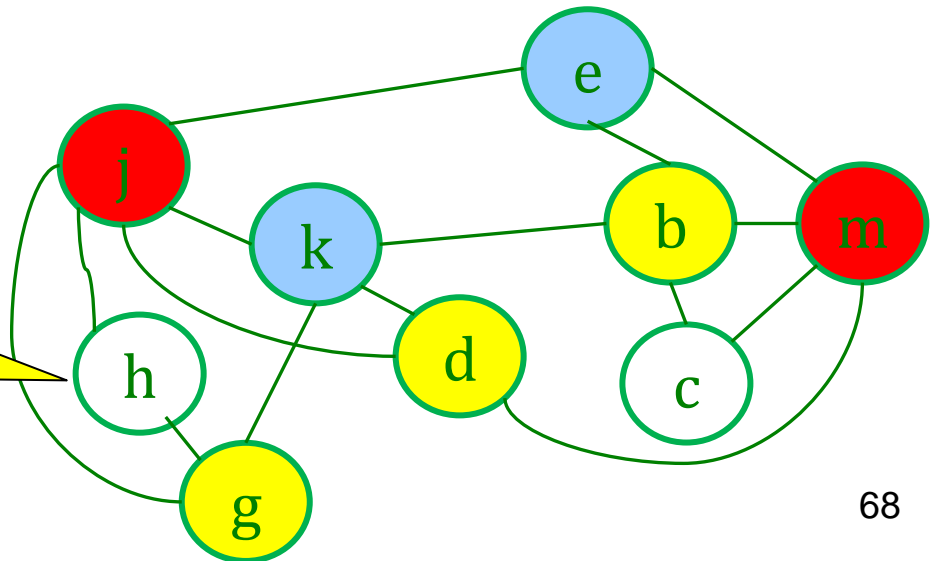
match S.choose (S.diff palette (colors\_of f (adj g n))) with

| Some c => M.add n c f

| None => f

end.

Find a color for this node that's not already used in an adjacent node

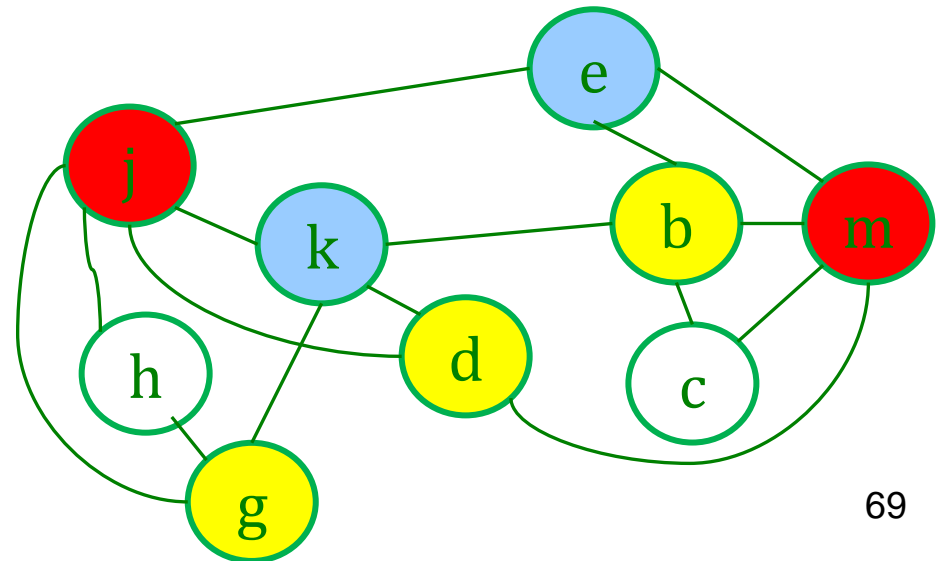


# The entire program

Definition `color (palette: S.t) (G: graph) : coloring :=`  
`fold_right (color1 palette G) (M.empty _) (select (S.cardinal palette) G).`

Phase 2

Phase 1



# PROVING THE PROGRAM CORRECT

# Specification of correctness

coloring : Type := “finite function from node to color”

Definition color (palette: S.t) (G: graph) : coloring :=

Correctness:

If  $f$  is a coloring for  $G$ , that is,  $\text{color palette } G = f$ ,

then (1) if  $f(i) = \text{Some } c$  then  $c \in \text{palette}$

and (2) if  $j \in G(i)$  and  $f(i) = \text{Some } c$  and  $f(j) = \text{Some } d$  then  $c \neq d$

Definition coloring\_ok (palette: S.t) (g: graph) (f: coloring) :=

forall i j, S.In j (adj g i) →

(forall ci, M.find i f = Some ci → S.In ci palette) /\

(forall ci cj, M.find i f = Some ci → M.find j f = Some cj → ci <> cj).

# Theorem

Definition coloring\_ok (palette: S.t) (g: graph) (f: coloring) :=  
 $\forall i j,$   
S.In j (adj g i)  $\rightarrow$   
(forall ci, M.find i f = Some ci  $\rightarrow$  S.In ci palette) /\  
(forall ci cj, M.find i f = Some ci  $\rightarrow$  M.find j f = Some cj  $\rightarrow$  ci <> cj).

Theorem color\_correct:

$\forall$  palette g,  
no\_selfloop g  $\rightarrow$   
undirected g  $\rightarrow$   
coloring\_ok palette g (color palette g).

**Proof: See the Coq development, Color.v**