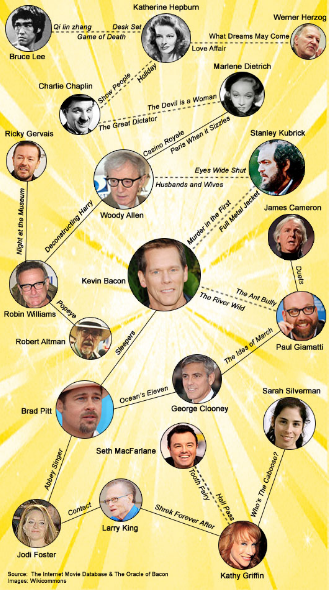
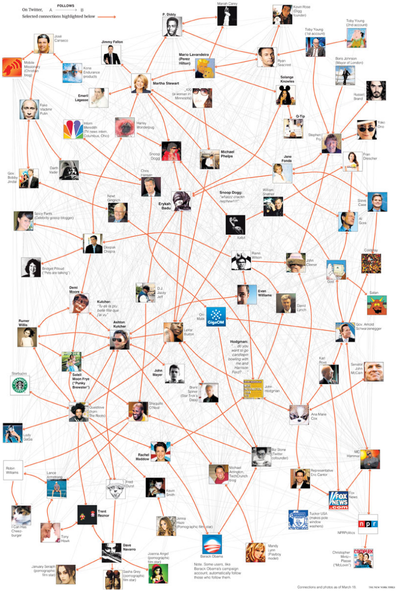
# Graphs

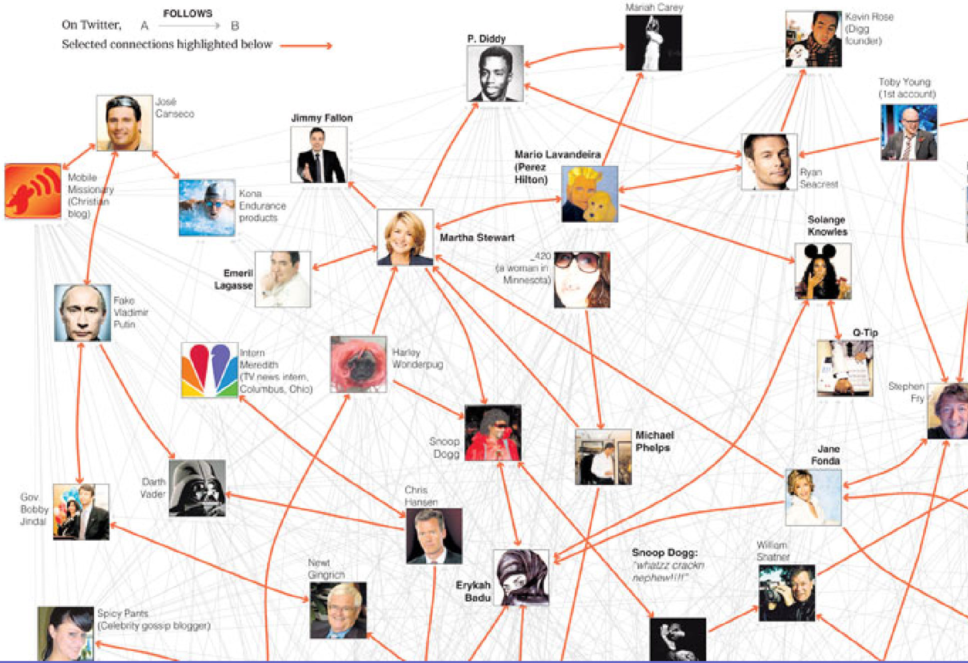
T. M. Murali

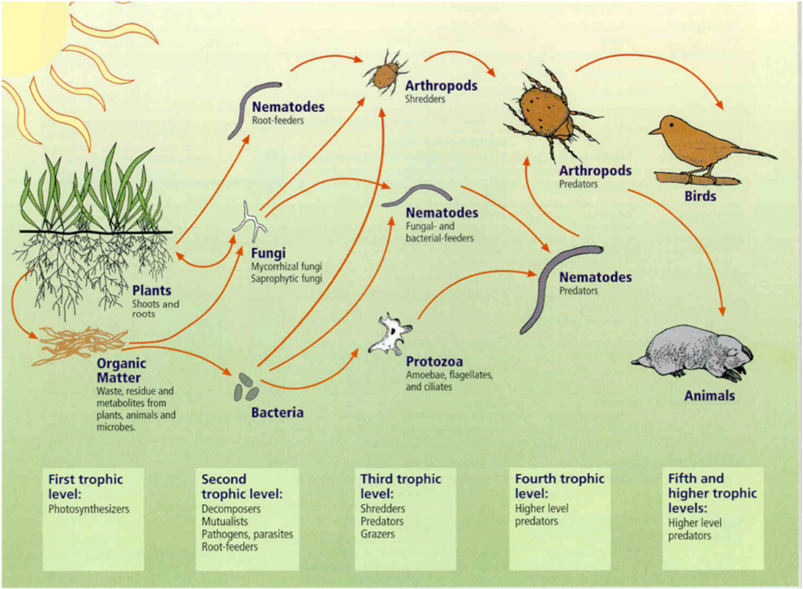
January 28, February 2, 4, 2016

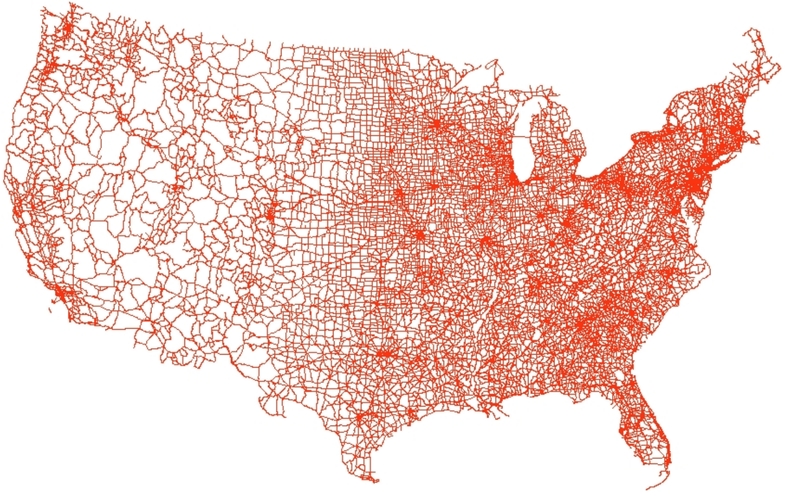


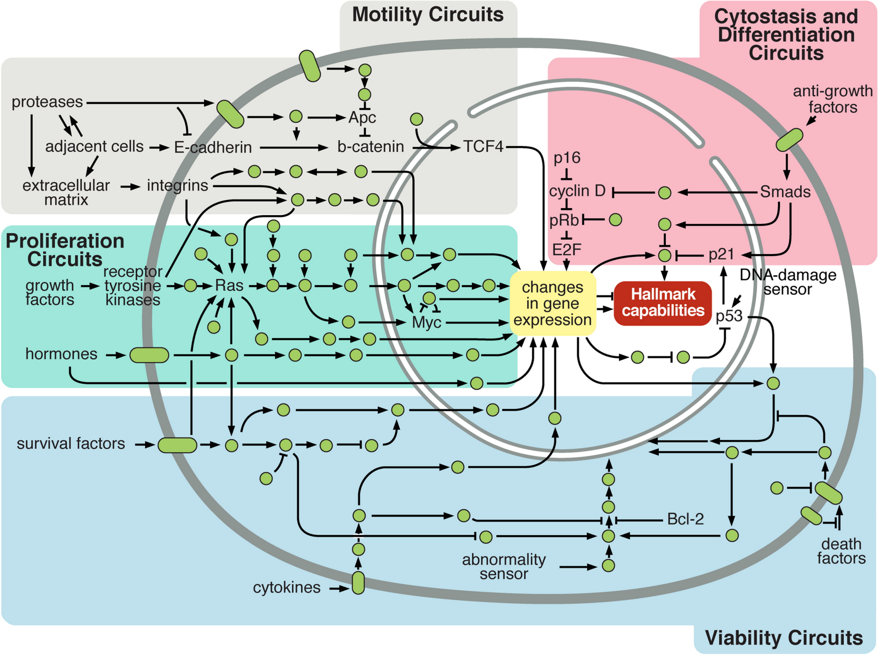

The Oracle of Bacon




On Twitter, A FOLLOWS B
Selected connections highlighted below
Mariah Carey
Kevin Rose (Digg founder)
P. Diddy
Toby Young (1st account)
José Canseco
Jimmy Fallon
Mario Lavandeira (Perez Hilton)
Ryan Seacrest
Mobile Missionary (Christian blog)
Kona Endurance products
Martha Stewart
Solange Knowles
_420 (a woman in Minnesota)
Emeril Lagasse
Fake Vladimir Putin
Q-Tip
Intern Meredith (TV news intern, Columbus, Ohio)
Harley Wonderpug
Stephen Fry
Snoop Dogg
Michael Phelps
Jane Fonda
Darth Vader
Gov. Bobby Jindal
Chris Hansen
Snoop Dogg: "whatzz crackn nephew!!!!"
William Shatner
Erykah Badu
Newt Gingrich
Spicy Pants (Celebrity gossip blogger)


Nematodes
Root-feeders
Arthropods
Shredders
Arthropods
Predators
Birds
Nematodes
Fungal- and
bacterial-feeders
Fungi
Mycorrhizal fungi
Saprophytic fungi
Plants
Shoots and
roots
Nematodes
Predators
Protozoa
Amoebae, flagellates,
and ciliates
Organic
Matter
Waste, residue and
metabolites from
plants, animals and
microbes.
Bacteria
Animals
First trophic
level:
Photosynthesizers
Second
trophic level:
Decomposers
Mutualists
Pathogens, parasites
Root-feeders
Third trophic
level:
Shredders
Predators
Grazers
Fourth trophic
level:
Higher level
predators
Fifth and
higher trophic
levels:
Higher level
predators




Motility Circuits
Cytostasis and Differentiation Circuits
Proliferation Circuits
Viability Circuits
proteases
adjacent cells
E-cadherin
extracellular matrix
integrins
Apc
b-catenin
TCF4
p16
cyclin D
pRb
E2F
anti-growth factors
Smads
p21
DNA-damage sensor
p53
growth factors
receptor tyrosine kinases
Ras
Myc
hormones
changes in gene expression
Hallmark capabilities
survival factors
Bcl-2
abnormality sensor
cytokines
death factors

# Graphs

- Model pairwise relationships (edges) between objects (nodes).

# Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications:

# Graphs

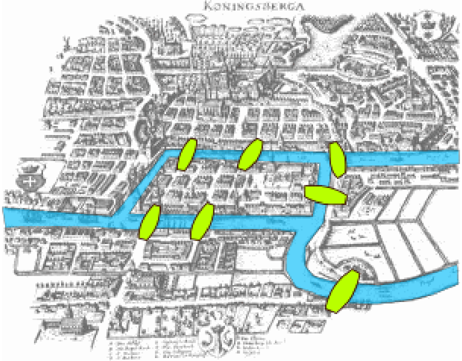
- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications: computer networks, the World Wide Web, ecology (food webs), social networks, software systems, job scheduling, VLSI circuits, cellular networks, ...
- Other examples: gene and protein networks, our bodies (nervous and circulatory systems, brains), buildings, transportation networks, ...

# Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications: computer networks, the World Wide Web, ecology (food webs), social networks, software systems, job scheduling, VLSI circuits, cellular networks, ...
- Other examples: gene and protein networks, our bodies (nervous and circulatory systems, brains), buildings, transportation networks, ...
- Problems involving graphs have a rich history dating back to Euler.

# Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications: computer networks, the World Wide Web, ecology (food webs), social networks, software systems, job scheduling, VLSI circuits, cellular networks, ...
- Other examples: gene and protein networks, our bodies (nervous and circulatory systems, brains), buildings, transportation networks, ...
- Problems involving graphs have a rich history dating back to Euler.



# Definition of a Graph

- *Undirected graph* $G = (V, E)$: set $V$ of nodes and set $E$ of edges, where $E \subseteq V \times V$. Elements of $E$ are unordered pairs.
  - Abuse of notation: write an edge $e$ between nodes $u$ and $v$ as $e = (u, v)$ and not as $e = \{u, v\}$.
  - Say that edge $e$ is *incident* on $u$ and on $v$.
  - Exactly one edge between any pair of nodes.
  - $G$ contains no self loops.

# Definition of a Graph

- *Undirected graph* $G = (V, E)$: set $V$ of nodes and set $E$ of edges, where $E \subseteq V \times V$. Elements of $E$ are unordered pairs.
  - Abuse of notation: write an edge $e$ between nodes $u$ and $v$ as $e = (u, v)$ and not as $e = \{u, v\}$.
  - Say that edge $e$ is *incident* on $u$ and on $v$.
  - Exactly one edge between any pair of nodes.
  - $G$ contains no self loops.
- *Directed graph* $G = (V, E)$: set $V$ of nodes and set $E$ of edges, where $E \subseteq V \times V$. Elements of $E$ are ordered pairs.

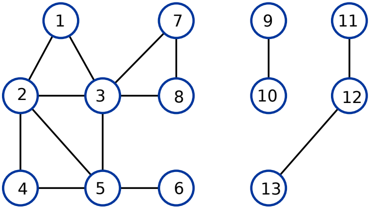
# Definition of a Graph

- *Undirected graph* $G = (V, E)$: set $V$ of nodes and set $E$ of edges, where $E \subseteq V \times V$. Elements of $E$ are unordered pairs.
  - Abuse of notation: write an edge $e$ between nodes $u$ and $v$ as $e = (u, v)$ and not as $e = \{u, v\}$.
  - Say that edge $e$ is *incident* on $u$ and on $v$.
  - Exactly one edge between any pair of nodes.
  - $G$ contains no self loops.
- *Directed graph* $G = (V, E)$: set $V$ of nodes and set $E$ of edges, where $E \subseteq V \times V$. Elements of $E$ are ordered pairs.
  - $e = (u, v)$: $u$ is the *tail* of the edge $e$, $v$ is its *head*; $e$ *leaves* node $u$ and *enters* node $v$; $e$ is *directed from $u$ to $v$*.
  - A pair of nodes $\{u, v\}$ may be connected by two directed edges: $(u, v)$ and $(v, u)$.
  - $G$ contains no self loops.

# Definition of a Graph

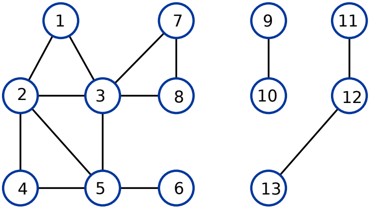
- *Undirected graph* $G = (V, E)$: set $V$ of nodes and set $E$ of edges, where $E \subseteq V \times V$. Elements of $E$ are unordered pairs.
  - Abuse of notation: write an edge $e$ between nodes $u$ and $v$ as $e = (u, v)$ and not as $e = \{u, v\}$.
  - Say that edge $e$ is *incident* on $u$ and on $v$.
  - Exactly one edge between any pair of nodes.
  - $G$ contains no self loops.
- *Directed graph* $G = (V, E)$: set $V$ of nodes and set $E$ of edges, where $E \subseteq V \times V$. Elements of $E$ are ordered pairs.
  - $e = (u, v)$: $u$ is the *tail* of the edge $e$, $v$ is its *head*; $e$ *leaves* node $u$ and *enters* node $v$; $e$ is *directed from $u$ to $v$*.
  - A pair of nodes $\{u, v\}$ may be connected by two directed edges: $(u, v)$ and $(v, u)$.
  - $G$ contains no self loops.
- By default, "graph" will mean an "undirected graph".

# Paths and Connectivity



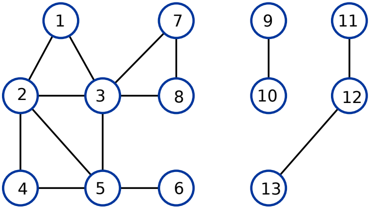

- A *path* in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in $E$.
  - $P$ is called a path *from $v_1$ to $v_K$* or a *$v_1$-$v_k$* path.
- A path is *simple* if all its nodes are distinct.
- A *cycle* is a path where $k > 2$, the first $k - 1$ nodes are distinct, and $v_1 = v_k$.

# Paths and Connectivity



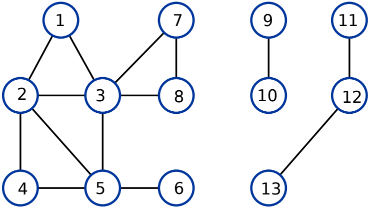

- A *path* in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in $E$.
  - $P$ is called a path *from $v_1$ to $v_K$* or a *$v_1$-$v_k$* path.
- A path is *simple* if all its nodes are distinct.
- A *cycle* is a path where $k > 2$, the first $k-1$ nodes are distinct, and $v_1 = v_k$.
  - All definitions carry over to directed graphs as well.

# Paths and Connectivity




- A *path* in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in $E$.
  - $P$ is called a path *from $v_1$ to $v_K$* or a *$v_1$-$v_k$* path.
- A path is *simple* if all its nodes are distinct.
- A *cycle* is a path where $k > 2$, the first $k - 1$ nodes are distinct, and $v_1 = v_k$.
  - All definitions carry over to directed graphs as well.
- An undirected graph $G$ is *connected* if for every pair of nodes $u, v \in V$, there is a path from $u$ to $v$ in $G$.
  - Directed graphs have the notion of "strong connectivity."

# Paths and Connectivity




- A *path* in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in $E$.
  - $P$ is called a path *from $v_1$ to $v_K$* or a *$v_1$-$v_k$* path.
- A path is *simple* if all its nodes are distinct.
- A *cycle* is a path where $k > 2$, the first $k - 1$ nodes are distinct, and $v_1 = v_k$.
  - All definitions carry over to directed graphs as well.
- An undirected graph $G$ is *connected* if for every pair of nodes $u, v \in V$, there is a path from $u$ to $v$ in $G$.
  - Directed graphs have the notion of "strong connectivity."
- *Distance* between two nodes $u$ and $v$ is the minimum number of edges in any $u$-$v$ path.

# Trees

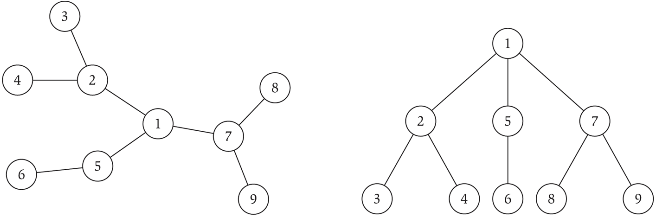



**Figure 3.1** Two drawings of the same tree. On the right, the tree is rooted at node 1.

- An undirected graph is a *tree* if it is connected and does not contain a cycle.

# Trees

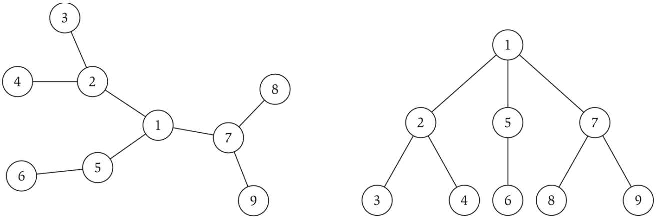



**Figure 3.1** Two drawings of the same tree. On the right, the tree is rooted at node 1.

- An undirected graph is a *tree* if it is connected and does not contain a cycle. For any pair of nodes in a tree, there is a unique path connecting them.

# Trees

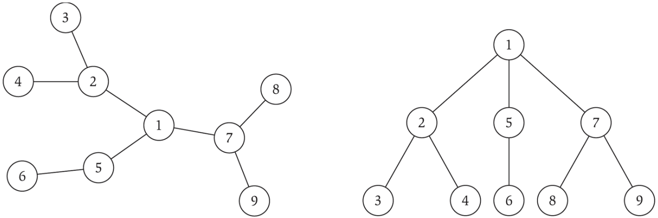



**Figure 3.1** Two drawings of the same tree. On the right, the tree is rooted at node 1.

- An undirected graph is a *tree* if it is connected and does not contain a cycle. For any pair of nodes in a tree, there is a unique path connecting them.
- *Rooting* a tree $T$: pick some node $r$ in the tree and orient each edge of $T$ "away" from $r$, i.e., for each node $v \neq r$, define *parent* of $v$ to be the node $u$ that directly precedes $v$ on the path from $r$ to $v$.

# Trees

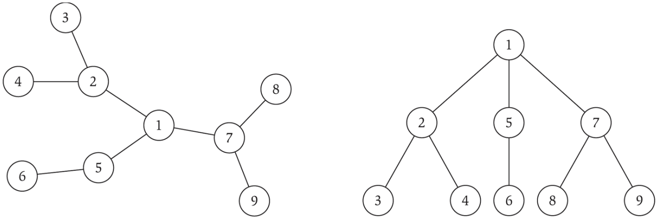



**Figure 3.1** Two drawings of the same tree. On the right, the tree is rooted at node 1.

- An undirected graph is a *tree* if it is connected and does not contain a cycle. For any pair of nodes in a tree, there is a unique path connecting them.
- *Rooting* a tree $T$: pick some node $r$ in the tree and orient each edge of $T$ "away" from $r$, i.e., for each node $v \neq r$, define *parent* of $v$ to be the node $u$ that directly precedes $v$ on the path from $r$ to $v$.
    - Node $w$ is a *child* of node $v$ if $v$ is a parent of $w$.
    - Node $w$ is a *descendant* of node $v$ (or $v$ is an *ancestor* of $w$) if $v$ lies on the $r$-$w$ path.
    - Node $x$ is a *leaf* if it has no descendants.

# Trees

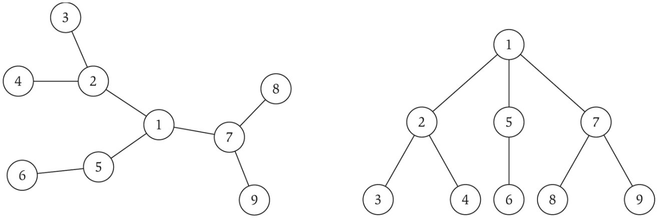



**Figure 3.1** Two drawings of the same tree. On the right, the tree is rooted at node 1.

- An undirected graph is a *tree* if it is connected and does not contain a cycle. For any pair of nodes in a tree, there is a unique path connecting them.
- *Rooting* a tree $T$: pick some node $r$ in the tree and orient each edge of $T$ "away" from $r$, i.e., for each node $v \neq r$, define *parent* of $v$ to be the node $u$ that directly precedes $v$ on the path from $r$ to $v$.
  - Node $w$ is a *child* of node $v$ if $v$ is a parent of $w$.
  - Node $w$ is a *descendant* of node $v$ (or $v$ is an *ancestor* of $w$) if $v$ lies on the $r$-$w$ path.
  - Node $x$ is a *leaf* if it has no descendants.
- Examples of (rooted) trees:

# Trees

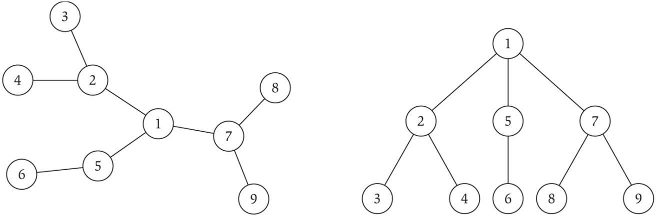



**Figure 3.1** Two drawings of the same tree. On the right, the tree is rooted at node 1.

- An undirected graph is a *tree* if it is connected and does not contain a cycle. For any pair of nodes in a tree, there is a unique path connecting them.
- *Rooting* a tree $T$: pick some node $r$ in the tree and orient each edge of $T$ "away" from $r$, i.e., for each node $v \neq r$, define *parent* of $v$ to be the node $u$ that directly precedes $v$ on the path from $r$ to $v$.
  - Node $w$ is a *child* of node $v$ if $v$ is a parent of $w$.
  - Node $w$ is a *descendant* of node $v$ (or $v$ is an *ancestor* of $w$) if $v$ lies on the $r$-$w$ path.
  - Node $x$ is a *leaf* if it has no descendants.
- Examples of (rooted) trees: organisational hierarchy, class hierarchies in object-oriented languages.

# Number of Edges in a Tree

- Claim: every $n$-node tree has edges.

# Number of Edges in a Tree

- Claim: every $n$-node tree has exactly $n-1$ edges.
- Proof 1:

# Number of Edges in a Tree

- Claim: every $n$-node tree has exactly $n-1$ edges.
- Proof 1: Root the tree. Each node, except the root, has a unique parent. Each edge connects one parent to one child. Therefore, the tree has $n-1$ edges.

# Number of Edges in a Tree

- Claim: every $n$-node tree has exactly $n - 1$ edges.
- Proof 1: Root the tree. Each node, except the root, has a unique parent. Each edge connects one parent to one child. Therefore, the tree has $n - 1$ edges.
- Proof 2: (by induction)

# Number of Edges in a Tree

- Claim: every $n$-node tree has exactly $n - 1$ edges.
- Proof 1: Root the tree. Each node, except the root, has a unique parent. Each edge connects one parent to one child. Therefore, the tree has $n - 1$ edges.
- Proof 2: (by induction) Two key pieces.
  - Every tree contains at least one leaf, i.e., node of degree 1. Why?
  - Inductive hypothesis: *every* tree with $n - 1$ nodes contains $n - 2$ edges.

# Number of Edges in a Tree

- Claim: every $n$-node tree has exactly $n-1$ edges.
- Proof 1: Root the tree. Each node, except the root, has a unique parent. Each edge connects one parent to one child. Therefore, the tree has $n-1$ edges.
- Proof 2: (by induction) Two key pieces.
  - Every tree contains at least one leaf, i.e., node of degree 1. Why?
  - Inductive hypothesis: *every* tree with $n-1$ nodes contains $n-2$ edges.
- Stronger claim: Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements implies the third:
  1. $G$ is connected.
  2. $G$ does not contain a cycle.
  3. $G$ contains $n-1$ edges.

# Number of Edges in a Tree

- Claim: every $n$-node tree has exactly $n-1$ edges.
- Proof 1: Root the tree. Each node, except the root, has a unique parent. Each edge connects one parent to one child. Therefore, the tree has $n-1$ edges.
- Proof 2: (by induction) Two key pieces.
  - Every tree contains at least one leaf, i.e., node of degree 1. Why?
  - Inductive hypothesis: *every* tree with $n-1$ nodes contains $n-2$ edges.
- Stronger claim: Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements implies the third:
  1. $G$ is connected.
  2. $G$ does not contain a cycle.
  3. $G$ contains $n-1$ edges.

  - Note that none of these statements uses the word "tree".

# Number of Edges in a Tree

- Claim: every $n$-node tree has exactly $n-1$ edges.
- Proof 1: Root the tree. Each node, except the root, has a unique parent. Each edge connects one parent to one child. Therefore, the tree has $n-1$ edges.
- Proof 2: (by induction) Two key pieces.
  - Every tree contains at least one leaf, i.e., node of degree 1. Why?
  - Inductive hypothesis: *every* tree with $n-1$ nodes contains $n-2$ edges.
- Stronger claim: Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements implies the third:
  1. $G$ is connected.
  2. $G$ does not contain a cycle.
  3. $G$ contains $n-1$ edges.

  - Note that none of these statements uses the word "tree".
  - 1 and 2 $\Rightarrow$ 3:

# Number of Edges in a Tree

- Claim: every $n$-node tree has exactly $n-1$ edges.
- Proof 1: Root the tree. Each node, except the root, has a unique parent. Each edge connects one parent to one child. Therefore, the tree has $n-1$ edges.
- Proof 2: (by induction) Two key pieces.
  - Every tree contains at least one leaf, i.e., node of degree 1. Why?
  - Inductive hypothesis: *every* tree with $n-1$ nodes contains $n-2$ edges.
- Stronger claim: Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements implies the third:
  1. $G$ is connected.
  2. $G$ does not contain a cycle.
  3. $G$ contains $n-1$ edges.

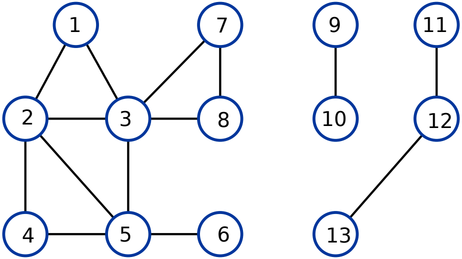
  - Note that none of these statements uses the word "tree".
  - 1 and 2 $\Rightarrow$ 3: just proved.
  - 2 and 3 $\Rightarrow$ 1:

# Number of Edges in a Tree

- Claim: every $n$-node tree has exactly $n-1$ edges.
- Proof 1: Root the tree. Each node, except the root, has a unique parent. Each edge connects one parent to one child. Therefore, the tree has $n-1$ edges.
- Proof 2: (by induction) Two key pieces.
  - Every tree contains at least one leaf, i.e., node of degree 1. Why?
  - Inductive hypothesis: *every* tree with $n-1$ nodes contains $n-2$ edges.
- Stronger claim: Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements implies the third:
  1. $G$ is connected.
  2. $G$ does not contain a cycle.
  3. $G$ contains $n-1$ edges.

  - Note that none of these statements uses the word "tree".
  - 1 and 2 $\Rightarrow$ 3: just proved.
  - 2 and 3 $\Rightarrow$ 1: prove by contradiction.
  - 3 and 1 $\Rightarrow$ 2: prove yourself.

# $s$-$t$ Connectivity

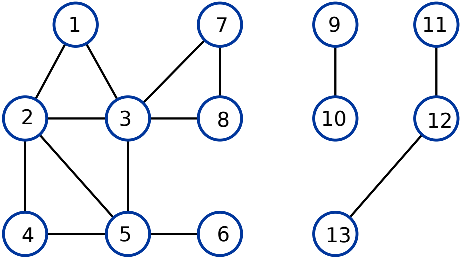



$s$-$t$ Connectivity

**INSTANCE:** An undirected graph $G = (V, E)$ and two nodes $s, t \in V$.

**QUESTION:** Is there an $s$-$t$ path in $G$?

# $s$-$t$ Connectivity



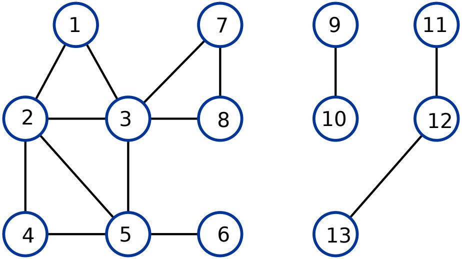

$s$-$t$ Connectivity

**INSTANCE:** An undirected graph $G = (V, E)$ and two nodes $s, t \in V$.

**QUESTION:** Is there an $s$-$t$ path in $G$?

- The *connected component of $G$ containing $s$* is the set of all nodes $u$ such that there is an $s$-$u$ path in $G$.

# $s$-$t$ Connectivity




$s$-$t$ Connectivity

**INSTANCE:** An undirected graph $G = (V, E)$ and two nodes $s, t \in V$.

**QUESTION:** Is there an $s$-$t$ path in $G$?

- The *connected component of $G$ containing $s$* is the set of all nodes $u$ such that there is an $s$-$u$ path in $G$.
- Algorithm for the $s$-$t$ Connectivity problem: compute the connected component of $G$ that contains $s$ and check if $t$ is in that component.

# Computing Connected Components

- "Explore" $G$ starting from $s$ and maintain set $R$ of visited nodes.

---

$R$ will consist of nodes to which $s$ has a path

Initially $R = \{s\}$

While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$

  Add $v$ to $R$

Endwhile

---

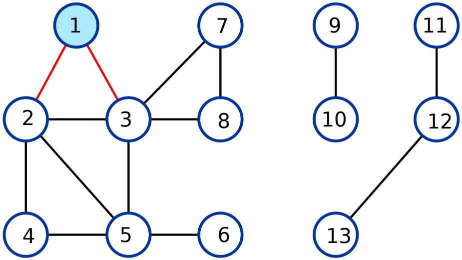
# Computing Connected Components

- "Explore" $G$ starting from $s$ and maintain set $R$ of visited nodes.

---

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$
Endwhile

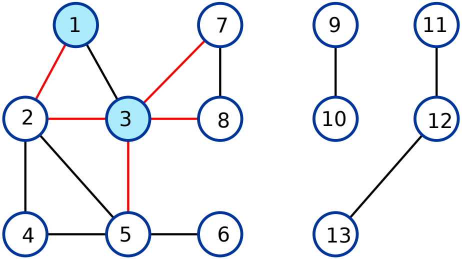
---



# Computing Connected Components

- "Explore" $G$ starting from $s$ and maintain set $R$ of visited nodes.

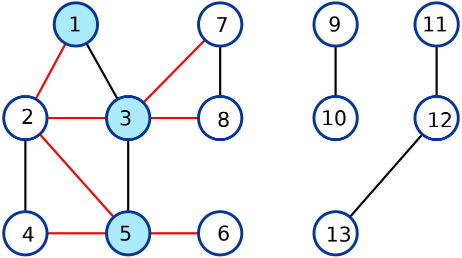
```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```



# Computing Connected Components

- "Explore" $G$ starting from $s$ and maintain set $R$ of visited nodes.

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```



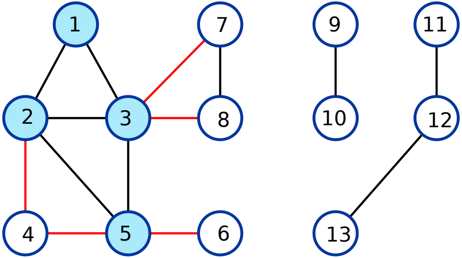
# Computing Connected Components

- "Explore" $G$ starting from $s$ and maintain set $R$ of visited nodes.

---

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```

---



# Computing Connected Components

- "Explore" $G$ starting from $s$ and maintain set $R$ of visited nodes.

---

$R$ will consist of nodes to which $s$ has a path

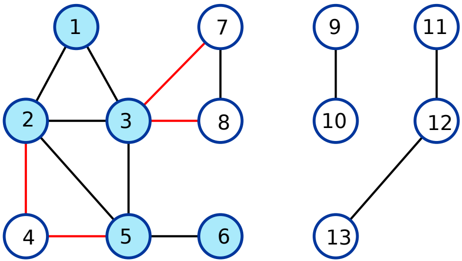
Initially $R = \{s\}$

While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$

  Add $v$ to $R$

Endwhile

---



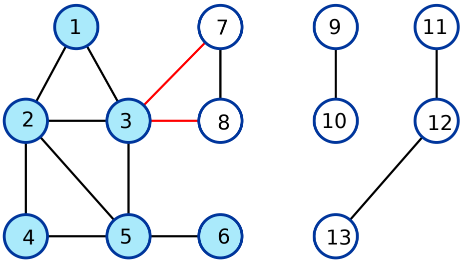
# Computing Connected Components

- "Explore" $G$ starting from $s$ and maintain set $R$ of visited nodes.

---

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$
Endwhile

---



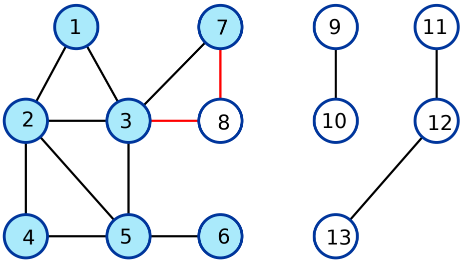
# Computing Connected Components

- "Explore" $G$ starting from $s$ and maintain set $R$ of visited nodes.

---

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```

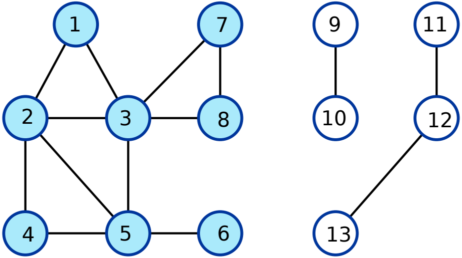
---



# Computing Connected Components

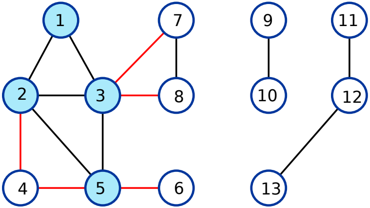
- "Explore" $G$ starting from $s$ and maintain set $R$ of visited nodes.

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```



# Issues in Computing Connected Components

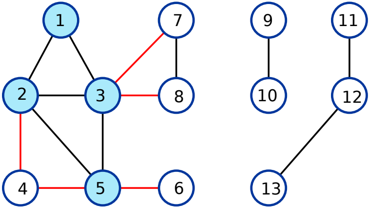
$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$
Endwhile




- How do we implement the `while` loop?

# Issues in Computing Connected Components

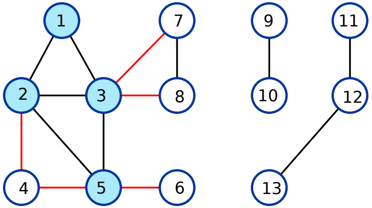
```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```




- How do we implement the `while` loop? Examine each edge in $E$.

# Issues in Computing Connected Components

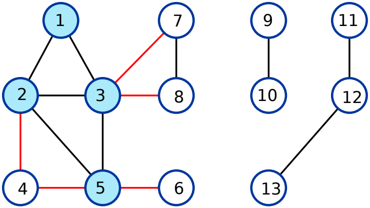
$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$
Endwhile




- How do we implement the `while` loop? Examine each edge in $E$.
- Other issues to consider:
  - Why does the algorithm terminate?
  - Does the algorithm truly compute connected component of $G$ containing $s$?
  - What is the running time of the algorithm?

# Termination of the Algorithm

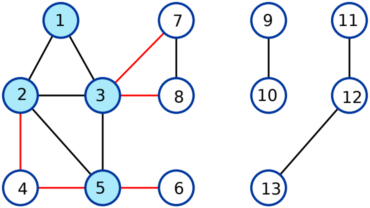
```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```




- How many nodes does each iteration of the `while` loop add to $R$?
- How many times is the while loop executed?

# Termination of the Algorithm

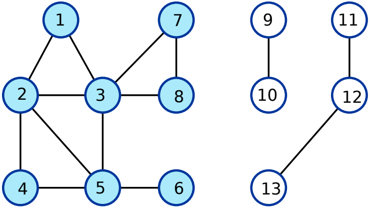
```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```




- How many nodes does each iteration of the `while` loop add to $R$? Exactly 1.
- How many times is the while loop executed?

# Termination of the Algorithm

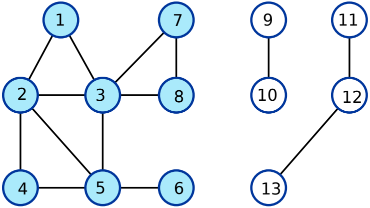
```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```




- How many nodes does each iteration of the `while` loop add to $R$? Exactly 1.
- How many times is the while loop executed? At most $n$ times.

# Termination of the Algorithm

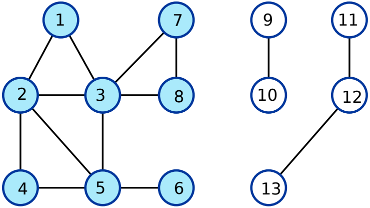
```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```




- How many nodes does each iteration of the `while` loop add to $R$? Exactly 1.
- How many times is the while loop executed? At most $n$ times.
- What is true of $R$ at termination?

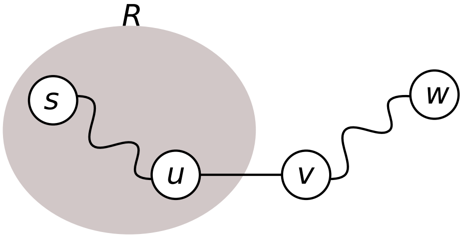
# Termination of the Algorithm

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$
Endwhile



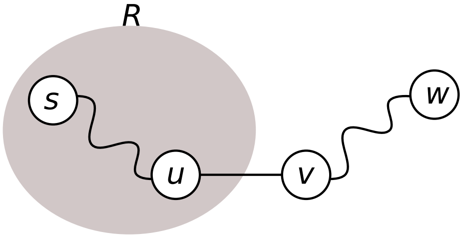

- How many nodes does each iteration of the `while` loop add to $R$? Exactly 1.
- How many times is the while loop executed? At most $n$ times.
- What is true of $R$ at termination?
  - either $R = V$ at the end or
  - in the last iteration, every edge either has both nodes in $R$ or both nodes not in $R$.

# Correctness of the Algorithm



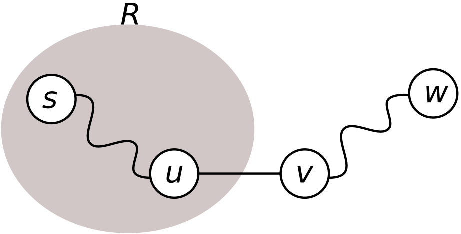

- Claim: at the end of the algorithm, the set $R$ is exactly the connected component of $G$ containing $s$.

# Correctness of the Algorithm



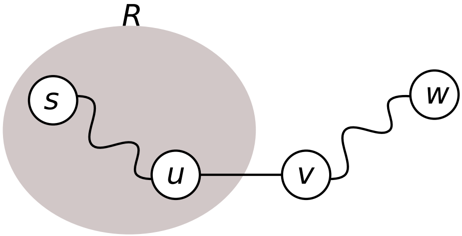

- Claim: at the end of the algorithm, the set $R$ is exactly the connected component of $G$ containing $s$.
- Proof: Suppose $w \notin R$ but there is an $s$-$w$ path $P$ in $G$.
  - Consider first node $v$ in $P$ not in $R$ ($v \neq s$).
  - Let $u$ be the predecessor of $v$ in $P$:

# Correctness of the Algorithm




- Claim: at the end of the algorithm, the set $R$ is exactly the connected component of $G$ containing $s$.
- Proof: Suppose $w \notin R$ but there is an $s$-$w$ path $P$ in $G$.
  - Consider first node $v$ in $P$ not in $R$ ($v \neq s$).
  - Let $u$ be the predecessor of $v$ in $P$: $u$ is in $R$.
  - $(u, v)$ is an edge with $u \in R$ but $v \notin R$, contradicting the stopping rule.

# Correctness of the Algorithm




- Claim: at the end of the algorithm, the set $R$ is exactly the connected component of $G$ containing $s$.
- Proof: Suppose $w \notin R$ but there is an $s$-$w$ path $P$ in $G$.
  - Consider first node $v$ in $P$ not in $R$ ($v \neq s$).
  - Let $u$ be the predecessor of $v$ in $P$: $u$ is in $R$.
  - $(u, v)$ is an edge with $u \in R$ but $v \notin R$, contradicting the stopping rule.
  - Note: wrong to assume that predecessor of $w$ in $P$ is not in $R$.

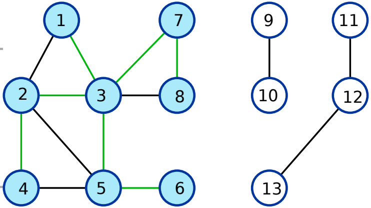
# Recovering Paths

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```

- Given a node $t \in R$, how do we recover the $s$-$t$ path?

# Recovering Paths

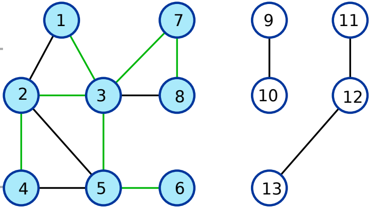
```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```




- Given a node $t \in R$, how do we recover the $s$-$t$ path?
- When adding node $v$ to $R$, record the edge $(u, v)$.
- What type of graph is formed by these edges?

# Recovering Paths

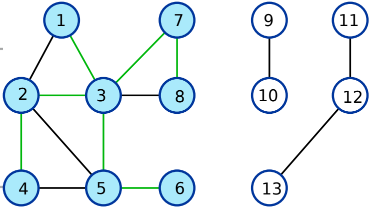
```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```




- Given a node $t \in R$, how do we recover the $s$-$t$ path?
- When adding node $v$ to $R$, record the edge $(u, v)$.
- What type of graph is formed by these edges? It is a tree! Why?

# Recovering Paths

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```




- Given a node $t \in R$, how do we recover the $s$-$t$ path?
- When adding node $v$ to $R$, record the edge $(u, v)$.
- What type of graph is formed by these edges? It is a tree! Why?
- To recover the $s$-$t$ path, trace these edges backwards from $t$ until we reach $s$.

# Running Time of the Algorithm

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```

# Running Time of the Algorithm

---

$R$ will consist of nodes to which $s$ has a path

Initially $R = \{s\}$

While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$

  Add $v$ to $R$

Endwhile

---

- Analyse algorithm in terms of two parameters: the number of nodes $n$ and the number of edges $m$.
- Implement the `while` loop by examining each edge in $E$. Running time of each loop is

# Running Time of the Algorithm

---

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$
Endwhile

---

- Analyse algorithm in terms of two parameters: the number of nodes $n$ and the number of edges $m$.
- Implement the `while` loop by examining each edge in $E$. Running time of each loop is $O(m)$.
- How many `while` loops does the algorithm execute?

# Running Time of the Algorithm

---

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$
Endwhile

---

- Analyse algorithm in terms of two parameters: the number of nodes $n$ and the number of edges $m$.
- Implement the `while` loop by examining each edge in $E$. Running time of each loop is $O(m)$.
- How many `while` loops does the algorithm execute? At most $n$.
- The running time is

# Running Time of the Algorithm

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```

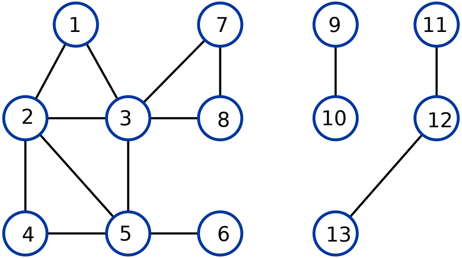
- Analyse algorithm in terms of two parameters: the number of nodes $n$ and the number of edges $m$.
- Implement the `while` loop by examining each edge in $E$. Running time of each loop is $O(m)$.
- How many `while` loops does the algorithm execute? At most $n$.
- The running time is $O(mn)$.

# Running Time of the Algorithm

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```

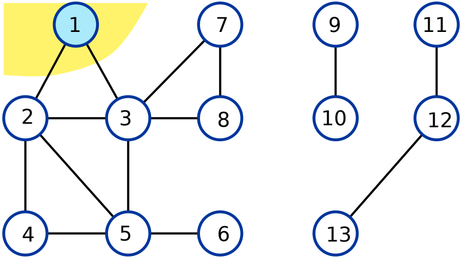
- Analyse algorithm in terms of two parameters: the number of nodes $n$ and the number of edges $m$.
- Implement the `while` loop by examining each edge in $E$. Running time of each loop is $O(m)$.
- How many `while` loops does the algorithm execute? At most $n$.
- The running time is $O(mn)$.
- Can we improve the running time by processing edges more carefully?

# Breadth-First Search (BFS)



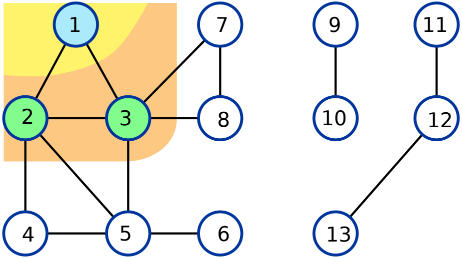

- Idea: explore $G$ starting at $s$ and going "outward" in all directions, adding nodes one layer at a time.

# Breadth-First Search (BFS)



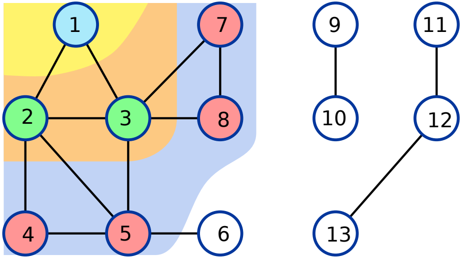

- Idea: explore $G$ starting at $s$ and going "outward" in all directions, adding nodes one layer at a time.
- Layer $L_0$ contains only $s$.

# Breadth-First Search (BFS)



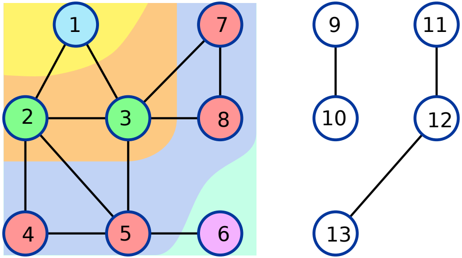

- Idea: explore $G$ starting at $s$ and going "outward" in all directions, adding nodes one layer at a time.
- Layer $L_0$ contains only $s$.
- Layer $L_1$ contains all neighbours of $s$.

# Breadth-First Search (BFS)



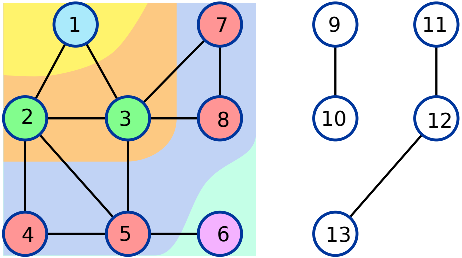

- Idea: explore $G$ starting at $s$ and going "outward" in all directions, adding nodes one layer at a time.
- Layer $L_0$ contains only $s$.
- Layer $L_1$ contains all neighbours of $s$.
- Given layers $L_0, L_1, \ldots, L_j$, layer $L_{j+1}$ contains all nodes that
  1. do not belong to an earlier layer and
  2. are connected by an edge to a node in layer $L_j$.

# Breadth-First Search (BFS)



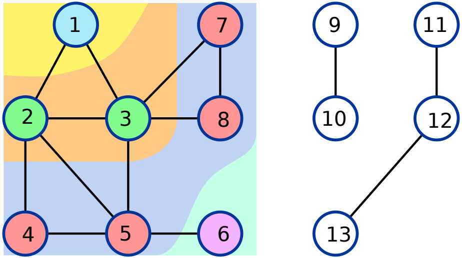

- Idea: explore $G$ starting at $s$ and going "outward" in all directions, adding nodes one layer at a time.
- Layer $L_0$ contains only $s$.
- Layer $L_1$ contains all neighbours of $s$.
- Given layers $L_0, L_1, \ldots, L_j$, layer $L_{j+1}$ contains all nodes that
  1. do not belong to an earlier layer and
  2. are connected by an edge to a node in layer $L_j$.

# Properties of BFS



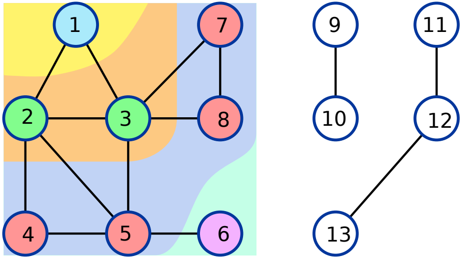

- We have not yet described how to compute these layers.
- Claim: For each $j \geq 1$, layer $L_j$ consists of all nodes

# Properties of BFS



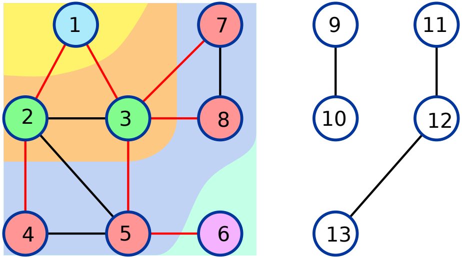

- We have not yet described how to compute these layers.
- Claim: For each $j \geq 1$, layer $L_j$ consists of all nodes exactly at distance $j$ from $S$. Proof

# Properties of BFS



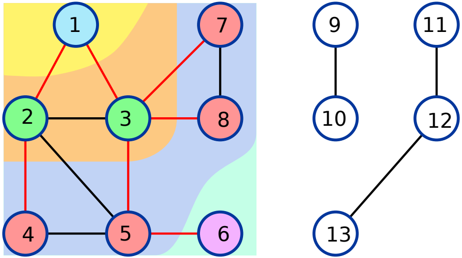

- We have not yet described how to compute these layers.
- Claim: For each $j \geq 1$, layer $L_j$ consists of all nodes exactly at distance $j$ from $S$. Proof by induction on $j$.
- Claim: There is a path from $s$ to $t$ if and only if $t$ is a member of some layer.

# Properties of BFS



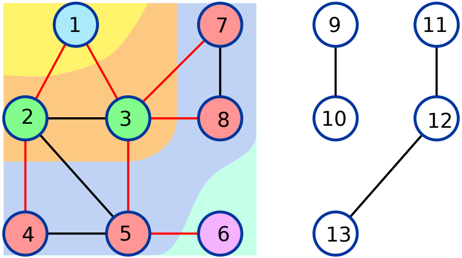

- We have not yet described how to compute these layers.
- Claim: For each $j \geq 1$, layer $L_j$ consists of all nodes exactly at distance $j$ from $S$. Proof by induction on $j$.
- Claim: There is a path from $s$ to $t$ if and only if $t$ is a member of some layer.
- Let $v$ be a node in layer $L_{j+1}$ and $u$ be the "first" node in $L_j$ such that $(u, v)$ is an edge in $G$. Consider the graph $T$ formed by all such edges, directed from $u$ to $v$.

# Properties of BFS



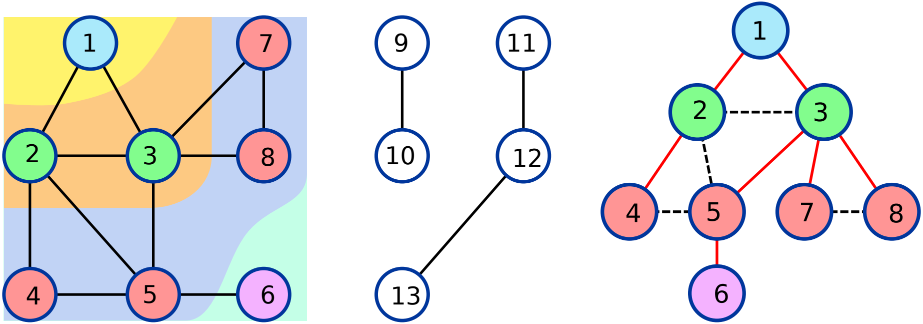

- We have not yet described how to compute these layers.
- Claim: For each $j \geq 1$, layer $L_j$ consists of all nodes exactly at distance $j$ from $S$. Proof by induction on $j$.
- Claim: There is a path from $s$ to $t$ if and only if $t$ is a member of some layer.
- Let $v$ be a node in layer $L_{j+1}$ and $u$ be the "first" node in $L_j$ such that $(u, v)$ is an edge in $G$. Consider the graph $T$ formed by all such edges, directed from $u$ to $v$.
  - Why is $T$ a tree?

# Properties of BFS



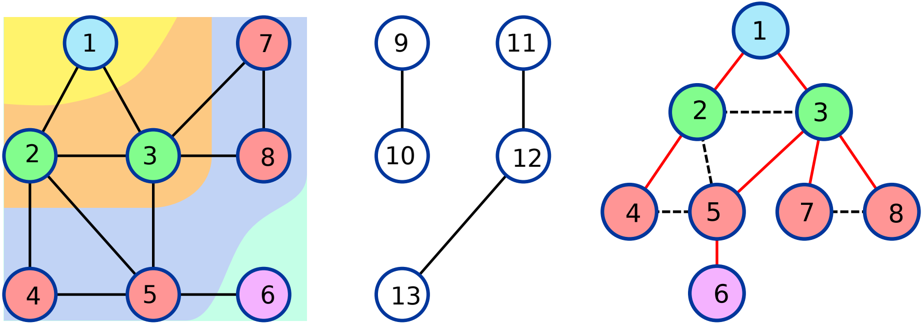

- We have not yet described how to compute these layers.
- Claim: For each $j \geq 1$, layer $L_j$ consists of all nodes exactly at distance $j$ from $S$. Proof by induction on $j$.
- Claim: There is a path from $s$ to $t$ if and only if $t$ is a member of some layer.
- Let $v$ be a node in layer $L_{j+1}$ and $u$ be the "first" node in $L_j$ such that $(u, v)$ is an edge in $G$. Consider the graph $T$ formed by all such edges, directed from $u$ to $v$.
  - Why is $T$ a tree? It is connected. The number of edges in $T$ is the number of nodes in all the layers minus 1.
  - $T$ is called the *breadth-first search tree*.

# BFS Trees



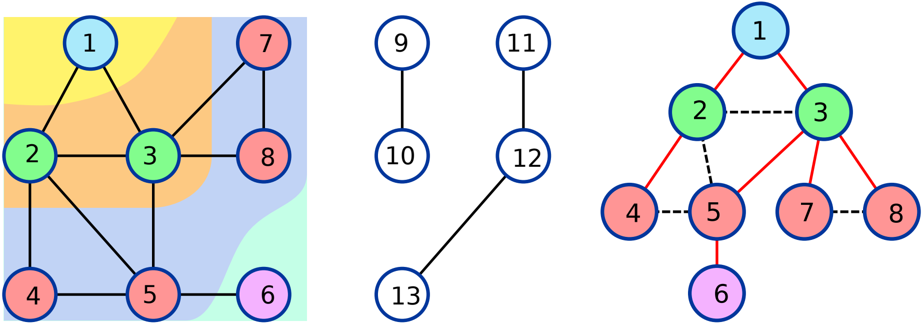

- *Non-tree edge*: an edge of $G$ that does not belong to the BFS tree $T$.
- Claim: Let $T$ be a BFS tree, let $x$ and $y$ be nodes in $T$ belonging to layers $L_i$ and $L_j$, respectively, and let $(x, y)$ be an edge of $G$. Then $|i - j| \leq 1$.

# BFS Trees




- *Non-tree edge*: an edge of $G$ that does not belong to the BFS tree $T$.
- Claim: Let $T$ be a BFS tree, let $x$ and $y$ be nodes in $T$ belonging to layers $L_i$ and $L_j$, respectively, and let $(x, y)$ be an edge of $G$. Then $|i - j| \leq 1$.
- Proof by contradiction: Suppose $i < j - 1$. Node $x \in L_i \Rightarrow$ all nodes adjacent to $x$ are in layers $L_1, L_2, \ldots L_{i+1}$. Hence $y$ must be in layer $L_{i+1}$ or earlier.

# BFS Trees




- *Non-tree edge*: an edge of $G$ that does not belong to the BFS tree $T$.
- Claim: Let $T$ be a BFS tree, let $x$ and $y$ be nodes in $T$ belonging to layers $L_i$ and $L_j$, respectively, and let $(x, y)$ be an edge of $G$. Then $|i - j| \leq 1$.
- Proof by contradiction: Suppose $i < j - 1$. Node $x \in L_i \Rightarrow$ all nodes adjacent to $x$ are in layers $L_1, L_2, \ldots L_{i+1}$. Hence $y$ must be in layer $L_{i+1}$ or earlier.
- Still unresolved: an efficient implementation of BFS.

# Depth-First Search (DFS)

- Explore $G$ as if it were a maze: start from $s$, traverse first edge out (to node $v$), traverse first edge out of $v$, . . . , reach a dead-end, backtrack, . . . . . .

# Depth-First Search (DFS)

- Explore $G$ as if it were a maze: start from $s$, traverse first edge out (to node $v$), traverse first edge out of $v$, ..., reach a dead-end, backtrack, ......

1. Mark all nodes as "`Unexplored`".
2. Invoke `DFS(`$s$`)`.

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u,v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

# Depth-First Search (DFS)

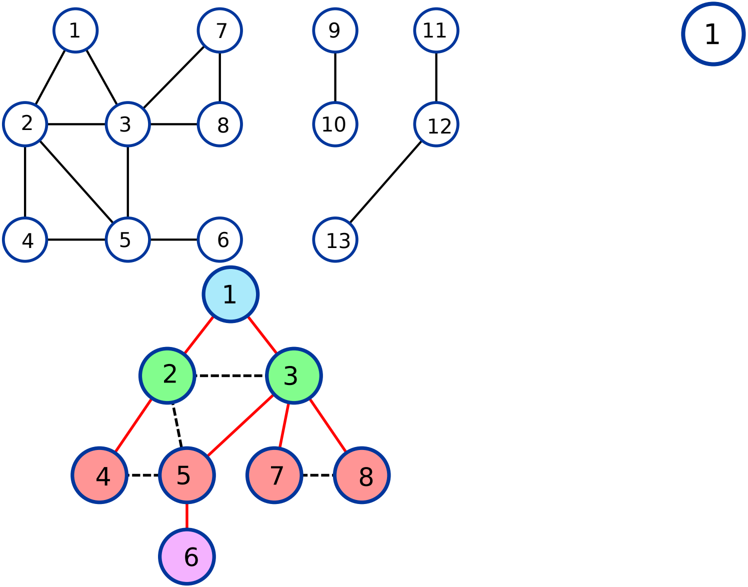
- Explore $G$ as if it were a maze: start from $s$, traverse first edge out (to node $v$), traverse first edge out of $v$, ..., reach a dead-end, backtrack, ......

1. Mark all nodes as "Unexplored".
2. Invoke DFS($s$).

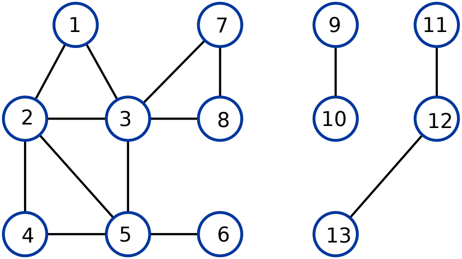
```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u, v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

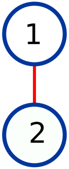
- *Depth-first search tree* is a tree $T$: when DFS($v$) is invoked directly during the call to DFS($v$), add edge $(u, v)$ to $T$.

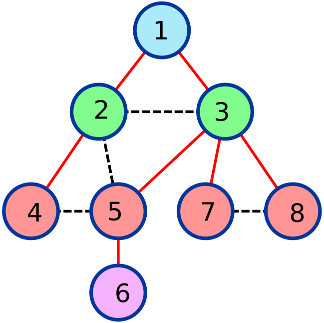
# Example of DFS



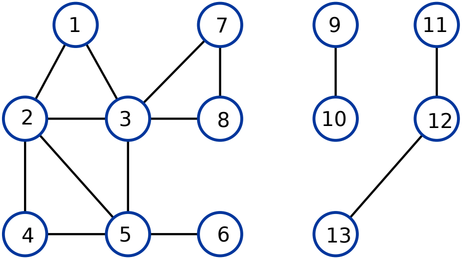
# Example of DFS

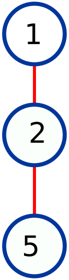


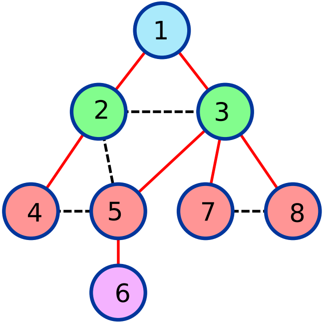



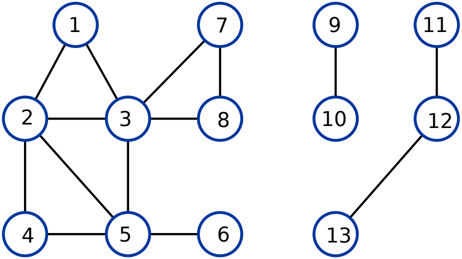
# Example of DFS

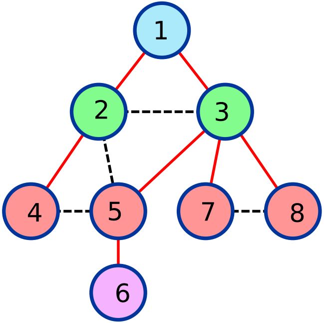


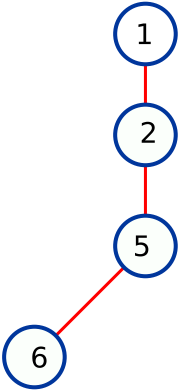



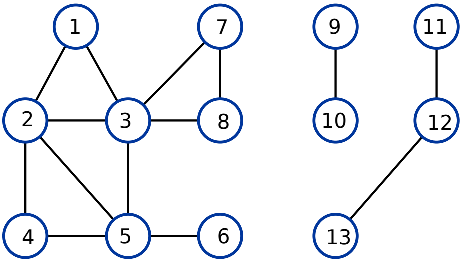
# Example of DFS

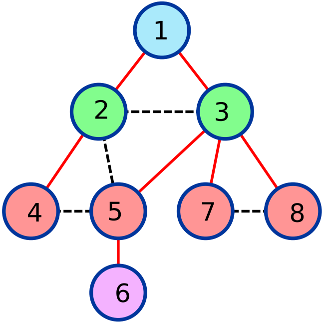


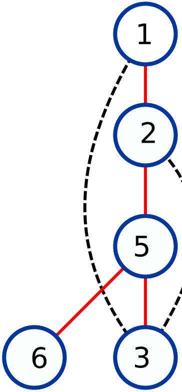



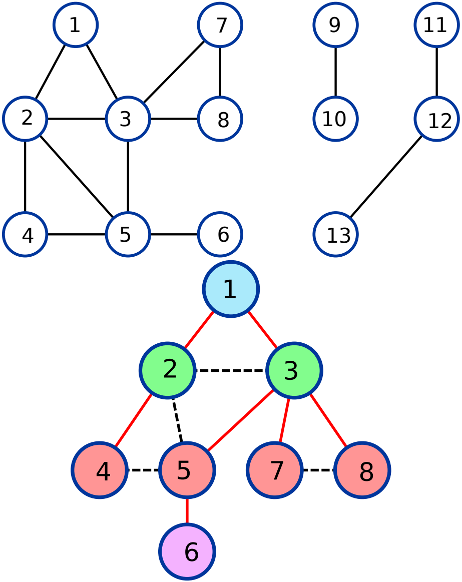
# Example of DFS

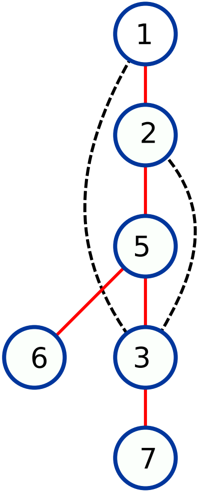


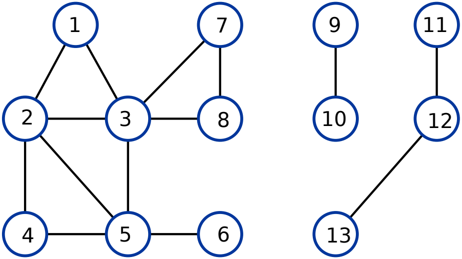



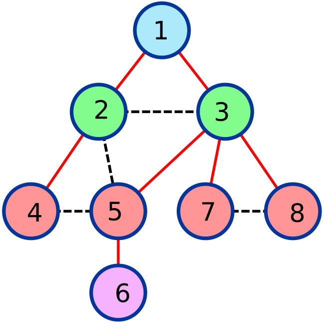
# Example of DFS

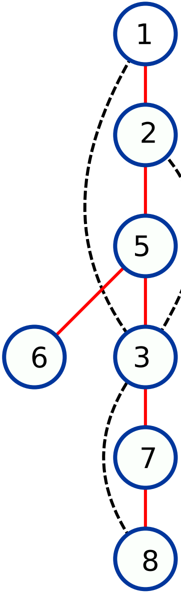


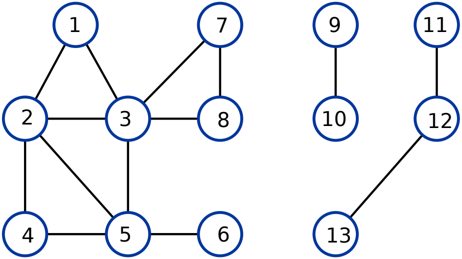
# Example of DFS

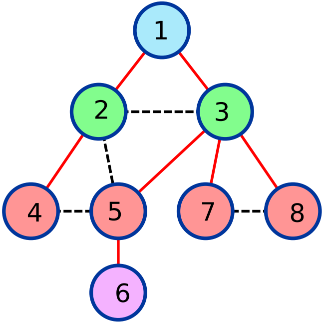


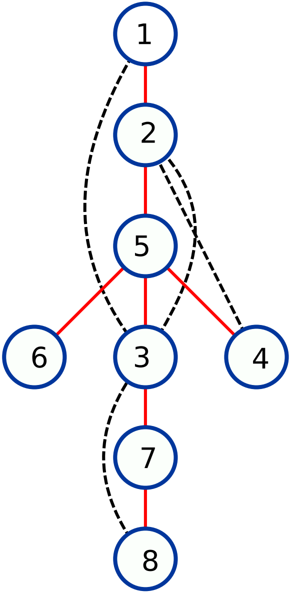



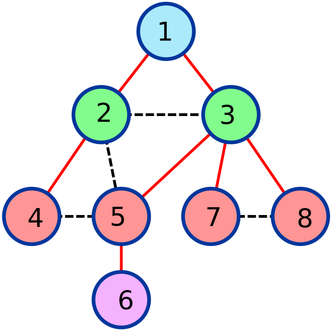
# Example of DFS

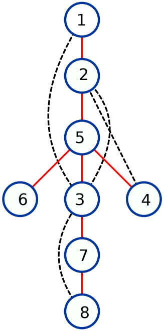


# BFS vs. DFS

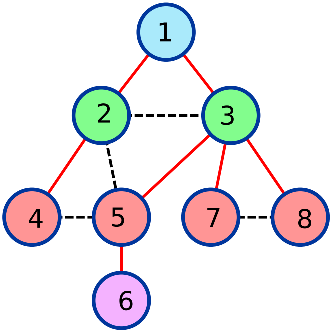


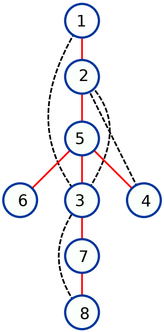




- Both visit the same set of nodes but in a different order.
- Both traverse all the edges in the connected component but in a different order.
- BFS trees have root-to-leaf paths that look as short as possible while paths in DFS trees tend to be long and deep.
- Non-tree edges

  BFS within the same level or between adjacent levels.

# BFS vs. DFS



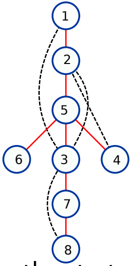

- Both visit the same set of nodes but in a different order.
- Both traverse all the edges in the connected component but in a different order.
- BFS trees have root-to-leaf paths that look as short as possible while paths in DFS trees tend to be long and deep.
- Non-tree edges

  BFS within the same level or between adjacent levels.

  DFS connect ancestors to descendants.

# Properties of DFS Trees

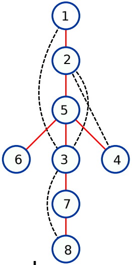
```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u, v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```




- Observation: All nodes marked as "Explored" between the start of DFS($u$) and its end are descendants of $u$ in the DFS tree $T$.

# Properties of DFS Trees

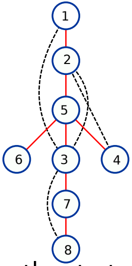
```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u,v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```




- Observation: All nodes marked as "Explored" between the start of DFS($u$) and its end are descendants of $u$ in the DFS tree $T$.
- Claim: Let $x$ and $y$ be nodes in a DFS tree $T$ such that $(x, y)$ is an edge of $G$ but not of $T$. Then one of $x$ or $y$ is an ancestor of the other in $T$.

# Properties of DFS Trees

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u,v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```




- Observation: All nodes marked as "Explored" between the start of DFS($u$) and its end are descendants of $u$ in the DFS tree $T$.
- Claim: Let $x$ and $y$ be nodes in a DFS tree $T$ such that $(x, y)$ is an edge of $G$ but not of $T$. Then one of $x$ or $y$ is an ancestor of the other in $T$.
- Proof: Assume, without loss of generality, that DFS($u$) reached $x$ first.
  - Since $(x, y)$ is an edge in $G$, it is examined during DFS($x$).
  - Since $(x, y) \notin T$, $y$ must be marked as "Explored" during DFS($x$) but before $(x, y)$ is examined.
  - Since $y$ was not marked as "Explored" before DFS($x$) was invoked, it must be marked as "Explored" between the start and the end of DFS($x$).
  - Therefore, $y$ must be a descendant of $x$ in $T$.

# All Connected Components

- We have discussed the component containing a particular node $s$.
- Each node belongs to a component.
- What is the relationship between all these components?

# All Connected Components

- We have discussed the component containing a particular node $s$.
- Each node belongs to a component.
- What is the relationship between all these components?
  - If $v$ is in $u$'s component, is $u$ in $v$'s component?
  - If $v$ is not in $u$'s component, can $u$ be in $v$'s component?

# All Connected Components

- We have discussed the component containing a particular node $s$.
- Each node belongs to a component.
- What is the relationship between all these components?
  - If $v$ is in $u$'s component, is $u$ in $v$'s component?
  - If $v$ is not in $u$'s component, can $u$ be in $v$'s component?
- Claim: For any two nodes $s$ and $t$ in a graph, their connected components are either equal or disjoint.

# All Connected Components

- We have discussed the component containing a particular node $s$.
- Each node belongs to a component.
- What is the relationship between all these components?
  - If $v$ is in $u$'s component, is $u$ in $v$'s component?
  - If $v$ is not in $u$'s component, can $u$ be in $v$'s component?
- Claim: For any two nodes $s$ and $t$ in a graph, their connected components are either equal or disjoint.
- Proof in two parts (sketch):
  1. If $G$ has an $s$-$t$ path, then the connected components of $s$ and $t$ are the same.

# All Connected Components

- We have discussed the component containing a particular node $s$.
- Each node belongs to a component.
- What is the relationship between all these components?
  - If $v$ is in $u$'s component, is $u$ in $v$'s component?
  - If $v$ is not in $u$'s component, can $u$ be in $v$'s component?
- Claim: For any two nodes $s$ and $t$ in a graph, their connected components are either equal or disjoint.
- Proof in two parts (sketch):
  1. If $G$ has an $s$-$t$ path, then the connected components of $s$ and $t$ are the same.
  2. If $G$ has no $s$-$t$ path, then there cannot be a node $v$ that is in both connected components.

# Computing All Connected Components

1. Pick an arbitrary node $s$ in $G$.
2. Compute its connected component using BFS (or DFS).
3. Find a node (say $v$, not already visited) and repeat the BFS from $v$.
4. Repeat this process until all nodes are visited.

# Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
    - Size of the graph is defined to be $m + n$.
    - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.

# Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is

# Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in

# Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n - 1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in

# Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in $\Theta(n)$ time.

# Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n - 1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in $\Theta(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to $v$.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.

# Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in $\Theta(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to $v$.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $n_v =$ the number of neighbours of node $v$.
  - Space used is

# Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in $\Theta(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to $v$.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $n_v =$ the number of neighbours of node $v$.
  - Space used is $O(n + \sum_{v \in G} n_v) =$

# Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in $\Theta(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to $v$.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $n_v =$ the number of neighbours of node $v$.
  - Space used is $O(n + \sum_{v \in G} n_v) = O(n + m)$, which is optimal for every graph.
  - Check if there is an edge between node $u$ and node $v$ in

# Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in $\Theta(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to $v$.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $n_v =$ the number of neighbours of node $v$.
  - Space used is $O(n + \sum_{v \in G} n_v) = O(n + m)$, which is optimal for every graph.
  - Check if there is an edge between node $u$ and node $v$ in $O(n_u)$ time.
  - Iterate over all the edges incident on node $u$ in

# Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in $\Theta(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to $v$.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $n_v =$ the number of neighbours of node $v$.
  - Space used is $O(n + \sum_{v \in G} n_v) = O(n + m)$, which is optimal for every graph.
  - Check if there is an edge between node $u$ and node $v$ in $O(n_u)$ time.
  - Iterate over all the edges incident on node $u$ in $\Theta(n_u)$ time.

# Data Structures for Implementation

- "Implementation" of BFS and DFS: fully specify the algorithms and data structures so that we can obtain provably efficient times.
- Inner loop of both BFS and DFS: process the set of edges incident on a given node and the set of visited nodes.
- How do we store the set of visited nodes? Order in which we process the nodes is crucial.

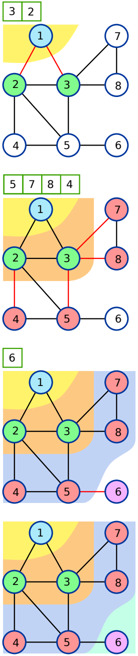
# Data Structures for Implementation

- "Implementation" of BFS and DFS: fully specify the algorithms and data structures so that we can obtain provably efficient times.
- Inner loop of both BFS and DFS: process the set of edges incident on a given node and the set of visited nodes.
- How do we store the set of visited nodes? Order in which we process the nodes is crucial.
  - BFS: store visited nodes in a queue (first-in, first-out).
  - DFS: store visited nodes in a stack (last-in, first-out)

# Implementing BFS

- Maintain an array `Discovered` and set `Discovered`$[v]$ = *true* as soon as the algorithm sees $v$.

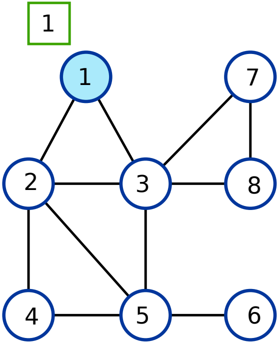
```
BFS(s):
  Set Discovered[s] = true and Discovered[v] = false for all other v
  Initialize L[0] to consist of the single element s
  Set the layer counter i = 0
  Set the current BFS tree T = Ø
  While L[i] is not empty
    Initialize an empty list L[i + 1]
    For each node u ∈ L[i]
      Consider each edge (u, v) incident to u
      If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Add v to the list L[i + 1]
      Endif
    Endfor
    Increment the layer counter i by one
  Endwhile
```



# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
      Pop the node u at the head of L
      Consider each edge (u, v) incident on u
      If Discovered[v] = false then
         Set Discovered[v] = true
         Add edge (u, v) to the tree T
         Push v to the back of L
      Endif
   Endwhile
```

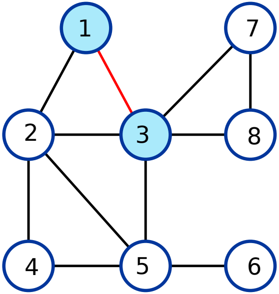


# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

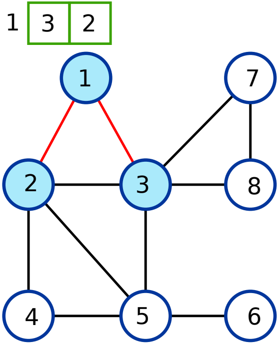
1 $\boxed{3}$



# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

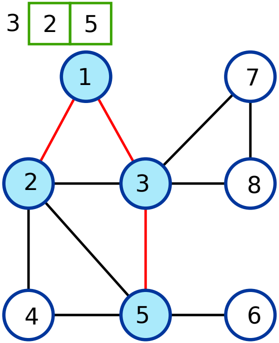
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```



# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

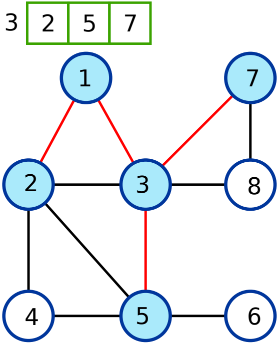
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
      Pop the node u at the head of L
      Consider each edge (u, v) incident on u
      If Discovered[v] = false then
         Set Discovered[v] = true
         Add edge (u, v) to the tree T
         Push v to the back of L
      Endif
   Endwhile
```



# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

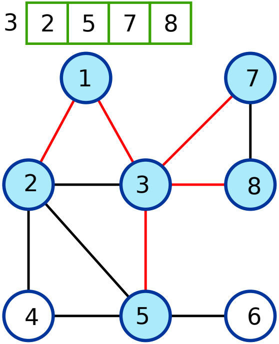
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
      Pop the node u at the head of L
      Consider each edge (u, v) incident on u
      If Discovered[v] = false then
         Set Discovered[v] = true
         Add edge (u, v) to the tree T
         Push v to the back of L
      Endif
   Endwhile
```



# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

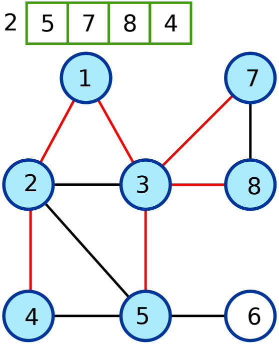
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
      Pop the node u at the head of L
      Consider each edge (u, v) incident on u
      If Discovered[v] = false then
         Set Discovered[v] = true
         Add edge (u, v) to the tree T
         Push v to the back of L
      Endif
   Endwhile
```



# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

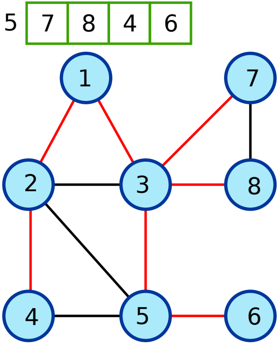
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```



# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

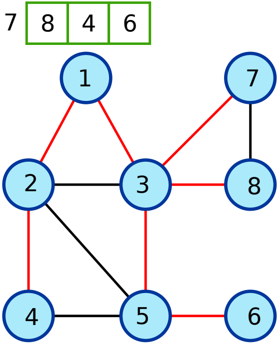
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```



# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

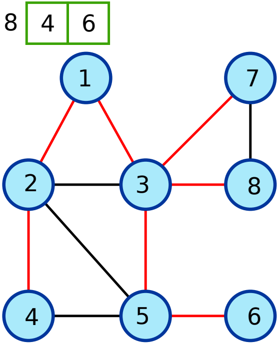
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```



# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

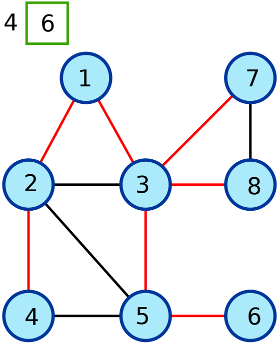
```
BFS(s):
  Set Discovered[s] = true
  Set Discovered[v] = false, for all other nodes v
  Initialize L to consist of the single element s
  While L is not empty
     Pop the node u at the head of L
     Consider each edge (u, v) incident on u
     If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Push v to the back of L
     Endif
  Endwhile
```



# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

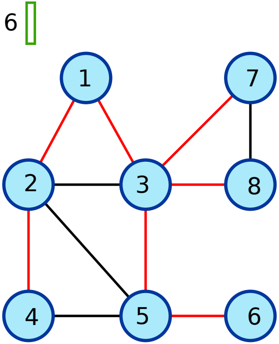
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
      Pop the node u at the head of L
      Consider each edge (u, v) incident on u
      If Discovered[v] = false then
         Set Discovered[v] = true
         Add edge (u, v) to the tree T
         Push v to the back of L
      Endif
   Endwhile
```



# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

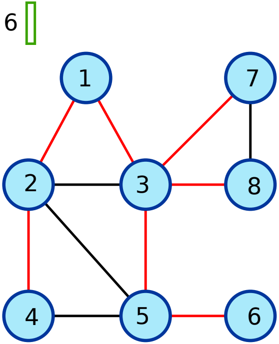
```
BFS(s):
  Set Discovered[s] = true
  Set Discovered[v] = false, for all other nodes v
  Initialize L to consist of the single element s
  While L is not empty
     Pop the node u at the head of L
     Consider each edge (u, v) incident on u
     If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Push v to the back of L
     Endif
  Endwhile
```



# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

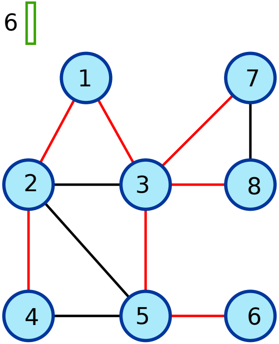



- Simple to modify this procedure to keep track of layer numbers as well.

# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
      Pop the node u at the head of L
      Consider each edge (u, v) incident on u
      If Discovered[v] = false then
         Set Discovered[v] = true
         Add edge (u, v) to the tree T
         Push v to the back of L
      Endif
   Endwhile
```

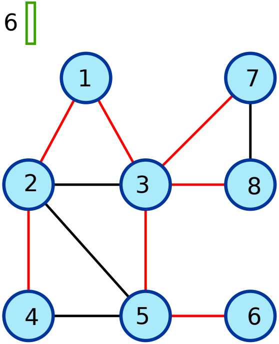



- Simple to modify this procedure to keep track of layer numbers as well. Store the pair $(u, l_u)$, where $l_u$ is the index of the layer containing $u$.

# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
      Pop the node u at the head of L
      Consider each edge (u, v) incident on u
      If Discovered[v] = false then
         Set Discovered[v] = true
         Add edge (u, v) to the tree T
         Push v to the back of L
      Endif
   Endwhile
```



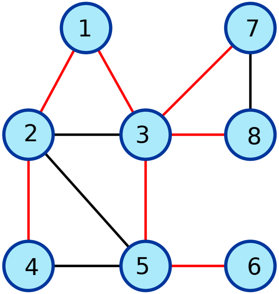

- Simple to modify this procedure to keep track of layer numbers as well. Store the pair $(u, l_u)$, where $l_u$ is the index of the layer containing $u$.
- Claim: Nodes in layer $i + 1$ will appear in $L$ immediately after nodes in layer $i$.

# Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue $L$.

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
      Pop the node u at the head of L
      Consider each edge (u, v) incident on u
      If Discovered[v] = false then
         Set Discovered[v] = true
         Add edge (u, v) to the tree T
         Push v to the back of L
      Endif
   Endwhile
```

6




- Simple to modify this procedure to keep track of layer numbers as well. Store the pair $(u, l_u)$, where $l_u$ is the index of the layer containing $u$.
- Claim: Nodes in layer $i+1$ will appear in $L$ immediately after nodes in layer $i$. More formally: If BFS($s$) pops $(v, l_v)$ from $L$ immediately after it pops $(u, l_u)$, then either $l_v = l_u$ or $l_v = l_u + 1$.

# Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- Naive bound on running time is

# Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- Naive bound on running time is $O(n^2)$: For each node, we spend $O(n)$ time.

# Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- Naive bound on running time is $O(n^2)$: For each node, we spend $O(n)$ time.
- Improved bound:
  - How many times is a node popped from $L$?

# Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- Naive bound on running time is $O(n^2)$: For each node, we spend $O(n)$ time.
- Improved bound:
  - How many times is a node popped from $L$? Exactly once.

# Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- Naive bound on running time is $O(n^2)$: For each node, we spend $O(n)$ time.
- Improved bound:
  - How many times is a node popped from $L$? Exactly once.
  - Time used by `for` loop for a node $u$:

# Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- Naive bound on running time is $O(n^2)$: For each node, we spend $O(n)$ time.
- Improved bound:
  - How many times is a node popped from $L$? Exactly once.
  - Time used by `for` loop for a node $u$: $O(n_u)$ time.

# Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- Naive bound on running time is $O(n^2)$: For each node, we spend $O(n)$ time.
- Improved bound:
  - How many times is a node popped from $L$? Exactly once.
  - Time used by `for` loop for a node $u$: $O(n_u)$ time.
  - Total time for all `for` loops: $\sum_{u \in G} O(n_u) = O(m)$ time.
  - Maintaining layer information:

# Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- Naive bound on running time is $O(n^2)$: For each node, we spend $O(n)$ time.
- Improved bound:
  - How many times is a node popped from $L$? Exactly once.
  - Time used by `for` loop for a node $u$: $O(n_u)$ time.
  - Total time for all `for` loops: $\sum_{u \in G} O(n_u) = O(m)$ time.
  - Maintaining layer information: $O(1)$ time per node.
  - Total time is $O(n + m)$.

# Recursive DFS

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u,v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

- Procedure has "tail recursion": recursive call is the last step.

# Recursive DFS

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u,v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

- Procedure has “tail recursion”: recursive call is the last step.
- Can replace the recursion by an iteration: use a stack to explicitly implement the recursion.

# Implementing DFS

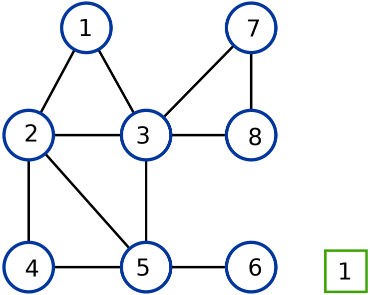
- Maintain a stack $S$ to store nodes to be explored.
- Maintain an array `Explored` and set $\texttt{Explored}[v] = \textit{true}$ when the algorithm pops $v$ from the stack.
- Read textbook on how to construct the DFS tree.

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u,v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```

# Implementing DFS

- Maintain a stack $S$ to store nodes to be explored.
- Maintain an array `Explored` and set `Explored`$[v]$ = *true* when the algorithm pops $v$ from the stack.
- Read textbook on how to construct the DFS tree.

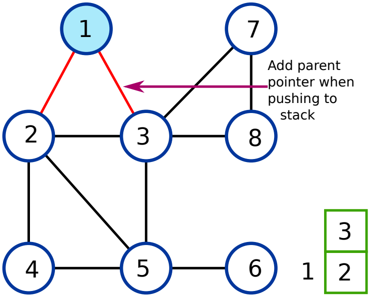
```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u,v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```



# Implementing DFS

- Maintain a stack $S$ to store nodes to be explored.
- Maintain an array `Explored` and set $\texttt{Explored}[v] = \textit{true}$ when the algorithm pops $v$ from the stack.
- Read textbook on how to construct the DFS tree.

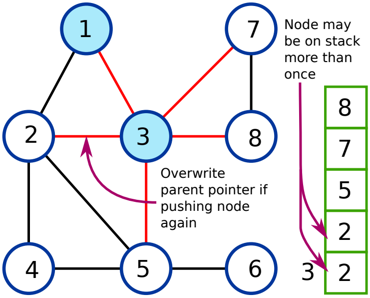
```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u,v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```



# Implementing DFS

- Maintain a stack $S$ to store nodes to be explored.
- Maintain an array `Explored` and set `Explored`$[v] = \textit{true}$ when the algorithm pops $v$ from the stack.
- Read textbook on how to construct the DFS tree.

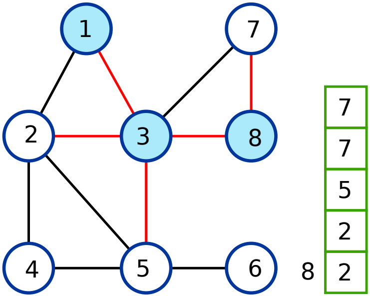
```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u, v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```



# Implementing DFS

- Maintain a stack $S$ to store nodes to be explored.
- Maintain an array Explored and set Explored$[v] = true$ when the algorithm pops $v$ from the stack.
- Read textbook on how to construct the DFS tree.

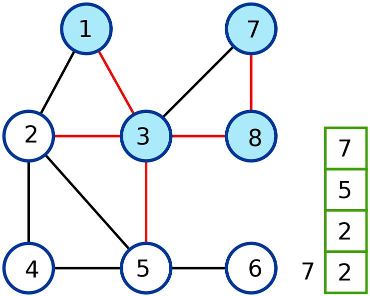
```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u,v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```



# Implementing DFS

- Maintain a stack $S$ to store nodes to be explored.
- Maintain an array `Explored` and set `Explored`$[v]$ = *true* when the algorithm pops $v$ from the stack.
- Read textbook on how to construct the DFS tree.

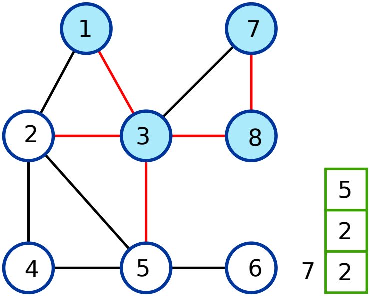
```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u, v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```



# Implementing DFS

- Maintain a stack $S$ to store nodes to be explored.
- Maintain an array `Explored` and set $\texttt{Explored}[v] = \textit{true}$ when the algorithm pops $v$ from the stack.
- Read textbook on how to construct the DFS tree.

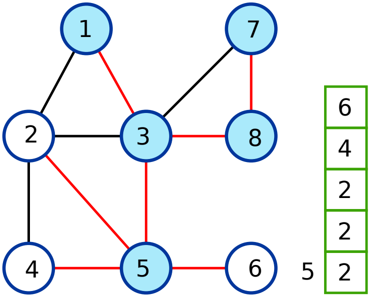
```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u, v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```



# Implementing DFS

- Maintain a stack $S$ to store nodes to be explored.
- Maintain an array `Explored` and set `Explored`$[v]$ = *true* when the algorithm pops $v$ from the stack.
- Read textbook on how to construct the DFS tree.

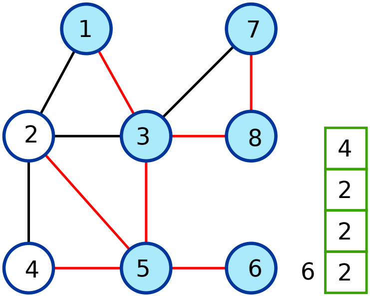
```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u, v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```



# Implementing DFS

- Maintain a stack $S$ to store nodes to be explored.
- Maintain an array `Explored` and set `Explored`$[v]$ = *true* when the algorithm pops $v$ from the stack.
- Read textbook on how to construct the DFS tree.

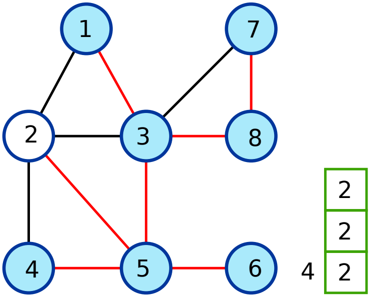
```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u,v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```



# Implementing DFS

- Maintain a stack $S$ to store nodes to be explored.
- Maintain an array `Explored` and set $\texttt{Explored}[v] = \textit{true}$ when the algorithm pops $v$ from the stack.
- Read textbook on how to construct the DFS tree.

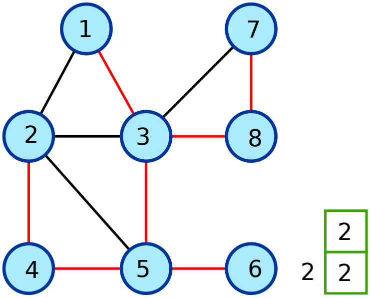
```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u,v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```



# Implementing DFS

- Maintain a stack $S$ to store nodes to be explored.
- Maintain an array `Explored` and set $\texttt{Explored}[v] = \textit{true}$ when the algorithm pops $v$ from the stack.
- Read textbook on how to construct the DFS tree.

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u, v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```



# Comparing Recursion and Iteration

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u,v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
       Set Explored[u] = true
       For each edge (u,v) incident to u
         Add v to the stack S
       Endfor
    Endif
  Endwhile
```

# Analysing DFS

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
       Set Explored[u] = true
       For each edge (u, v) incident to u
         Add v to the stack S
       Endfor
    Endif
  Endwhile
```

- How many times is a node's adjacency list scanned?

# Analysing DFS

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
       Set Explored[u] = true
       For each edge (u, v) incident to u
         Add v to the stack S
       Endfor
    Endif
  Endwhile
```

- How many times is a node's adjacency list scanned? Exactly once.

# Analysing DFS

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
       Set Explored[u] = true
       For each edge (u, v) incident to u
         Add v to the stack S
       Endfor
    Endif
  Endwhile
```

- How many times is a node's adjacency list scanned? Exactly once.
- The total amount of time to process edges incident on node $u$'s is

# Analysing DFS

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
       Set Explored[u] = true
       For each edge (u, v) incident to u
         Add v to the stack S
       Endfor
    Endif
  Endwhile
```

- How many times is a node's adjacency list scanned? Exactly once.
- The total amount of time to process edges incident on node $u$'s is $O(n_u)$.
- The total running time of the algorithm is

# Analysing DFS

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
       Set Explored[u] = true
       For each edge (u, v) incident to u
         Add v to the stack S
       Endfor
    Endif
  Endwhile
```

- How many times is a node's adjacency list scanned? Exactly once.
- The total amount of time to process edges incident on node $u$'s is $O(n_u)$.
- The total running time of the algorithm is $O(n + m)$.