Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Bridges in a Graph
Examples of Graphs

Figure: Road Networks

Figure: Social Networks

Figure: Internet

Figure: Transition Graphs
(Undirected) Graph $G = (V, E)$

- $V$: set of vertices (nodes);
- $E$: pairwise relationships among $V$;
  - (undirected) graphs: relationship is symmetric, $E$ contains subsets of size 2
(Undirected) Graph $G = (V, E)$

- $V$: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $E$: pairwise relationships among $V$;
  - (undirected) graphs: relationship is symmetric, $E$ contains subsets of size 2
  - $E = \\{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$
Directed Graph $G = (V, E)$

- $V$: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $E$: pairwise relationships among $V$;
  - **directed** graphs: relationship is asymmetric, $E$ contains ordered pairs
Directed Graph $G = (V, E)$

- $V$: set of vertices (nodes);
  
  $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$

- $E$: pairwise relationships among $V$;
  
  - directed graphs: relationship is asymmetric, $E$ contains ordered pairs
  
  $E = \{(1, 2), (1, 3), (3, 2), (4, 2), (2, 5), (5, 3), (3, 7), (3, 8), (4, 5), (5, 6), (6, 5), (8, 7)\}$
Abuse of Notations

- For (undirected) graphs, we often use \((i, j)\) to denote the set \(\{i, j\}\).
- We call \((i, j)\) an unordered pair; in this case \((i, j) = (j, i)\).

\[E = \{(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 7), (3, 8), (4, 5), (5, 6), (7, 8)\}\]
• Social Network : Undirected
• Transition Graph : Directed
• Road Network : Directed or Undirected
• Internet : Directed or Undirected
Adjacency matrix
- $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
- $A$ is symmetric if graph is undirected
Representation of Graphs

- **Adjacency matrix**
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - $A$ is symmetric if graph is undirected

- **Linked lists**
  - For every vertex $v$, there is a linked list containing all neighbours of $v$. 
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- $n$: number of vertices
- $m$: number of edges, assuming $n - 1 \leq m \leq n(n - 1)/2$
- $d_v$: number of neighbors of $v$

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>$O(n^2)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>time to check $(u, v) \in E$</td>
<td>$O(1)$</td>
<td>$O(d_u)$</td>
</tr>
<tr>
<td>time to list all neighbours of $v$</td>
<td>$O(n)$</td>
<td>$O(d_v)$</td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>( O(n^2) )</td>
<td></td>
</tr>
<tr>
<td>time to check ((u,v) \in E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time to list all neighbours of ( v )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Comparison of Two Representations**

- Assuming we are dealing with undirected graphs
- $n$: number of vertices
- $m$: number of edges, assuming $n - 1 \leq m \leq n(n - 1)/2$
- $d_v$: number of neighbors of $v$

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>$O(n^2)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>time to check $(u, v) \in E$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time to list all neighbours of $v$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>( O(n^2) )</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>time to check ( (u, v) \in E )</td>
<td>( O(1) )</td>
<td></td>
</tr>
<tr>
<td>time to list all neighbours of ( v )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>( O(n^2) )</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>time to check ( (u, v) \in E )</td>
<td>( O(1) )</td>
<td>( O(d_u) )</td>
</tr>
<tr>
<td>time to list all neighbours of ( v )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>( O(n^2) )</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>time to check ( u, v ) ( \in E )</td>
<td>( O(1) )</td>
<td>( O(d_u) )</td>
</tr>
<tr>
<td>time to list all neighbours of ( v )</td>
<td>( O(n) )</td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- $n$: number of vertices
- $m$: number of edges, assuming $n - 1 \leq m \leq n(n - 1)/2$
- $d_v$: number of neighbors of $v$

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>$O(n^2)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>time to check $(u, v) \in E$</td>
<td>$O(1)$</td>
<td>$O(d_u)$</td>
</tr>
<tr>
<td>time to list all neighbours of $v$</td>
<td>$O(n)$</td>
<td>$O(d_v)$</td>
</tr>
</tbody>
</table>
Outline

1. Graphs
2. Connectivity and Graph Traversal
   - Testing Bipartiteness
3. Topological Ordering
4. Bridges in a Graph
Connectivity Problem

**Input:** graph $G = (V, E)$, (using linked lists)
  
  two vertices $s, t \in V$

**Output:** whether there is a path connecting $s$ to $t$ in $G$
Connectivity Problem

**Input:** graph $G = (V, E)$, (using linked lists)
  two vertices $s, t \in V$

**Output:** whether there is a path connecting $s$ to $t$ in $G$

- Algorithm: starting from $s$, search for all vertices that are reachable from $s$ and check if the set contains $t$
Connectivity Problem

**Input:** graph $G = (V, E)$, (using linked lists)

two vertices $s, t \in V$

**Output:** whether there is a path connecting $s$ to $t$ in $G$

- Algorithm: starting from $s$, search for all vertices that are reachable from $s$ and check if the set contains $t$
- Breadth-First Search (BFS)
Connectivity Problem

**Input:** graph $G = (V, E)$, (using linked lists)

two vertices $s, t \in V$

**Output:** whether there is a path connecting $s$ to $t$ in $G$

- Algorithm: starting from $s$, search for all vertices that are reachable from $s$ and check if the set contains $t$
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \ldots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Implementing BFS using a Queue

BFS(s)

1. $head \leftarrow 1$, $tail \leftarrow 1$, $queue[1] \leftarrow s$
2. mark $s$ as “visited” and all other vertices as “unvisited”
3. while $head \geq tail$
4. $v \leftarrow queue[tail]$, $tail \leftarrow tail + 1$
5. for all neighbours $u$ of $v$
6. if $u$ is “unvisited” then
7. $head \leftarrow head + 1$, $queue[head] = u$
8. mark $u$ as “visited”

- Running time: $O(n + m)$. 
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue

Example of BFS via Queue
Example of BFS via Queue

A diagram illustrating the Breadth-First Search (BFS) algorithm using a queue. The queue is represented as a list of nodes with an arrow pointing from the head to the tail. The vertices in the graph are numbered, and the queue contains the order in which nodes are visited.
Example of BFS via Queue

```
1 2 3 4 5 7
```

```
head
```

```
tail
```
Example of BFS via Queue
Example of BFS via Queue

![Graph Example](image)

```
1 2 3 4 5 7 8
```

- **v**: Selected vertex
- **Queue**: 1, 2, 3, 4, 5, 7, 8
- **Head**: 1
- **Tail**: 8
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue

Diagram of a graph with nodes 1 through 7 and edges connecting them. The node marked with a red arrow is labeled as 'v'. To the right, a queue is shown with the nodes in the order 1, 2, 3, 4, 5, 7, 8, 6. The queue has an arrow pointing to the 'head' and another arrow pointing to the 'tail'.
Example of BFS via Queue
Example of BFS via Queue
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex (“dead-end”), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Implementing DFS using Recursion

**DFS(s)**

1. mark all vertices as “unvisited”
2. recursive-DFS(s)

**recursive-DFS(v)**

1. mark v as “visited”
2. for all neighbours u of v
3. if u is unvisited then recursive-DFS(u)
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Bridges in a Graph
Def. A graph $G = (V, E)$ is a bipartite graph if there is a partition of $V$ into two sets $L$ and $R$ such that for every edge $(u, v) \in E$, we have either $u \in L, v \in R$ or $v \in L, u \in R$. 
Testing Bipartiteness

Taking an arbitrary vertex \( s \in V \)

Assuming \( s \in L \) w.l.o.g. Neighbors of \( s \) must be in \( R \). Neighbors of neighbors of \( s \) must be in \( L \).···

Report "not a bipartite graph" if contradiction was found.

If \( G \) contains multiple connected components, repeat above algorithm for each component.
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of $s$ must be in $R$

Report "not a bipartite graph" if contradiction was found

If $G$ contains multiple connected components, repeat above algorithm for each component
Testing Bipartiteness

- Taking an arbitrary vertex \( s \in V \)
- Assuming \( s \in L \) w.l.o.g
- Neighbors of \( s \) must be in \( R \)
- Neighbors of neighbors of \( s \) must be in \( L \)

Report "not a bipartite graph" if contradiction was found

If \( G \) contains multiple connected components, repeat above algorithm for each component
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of $s$ must be in $R$
- Neighbors of neighbors of $s$ must be in $L$
- ...
Testing Bipartiteness

- Taking an arbitrary vertex \( s \in V \)
- Assuming \( s \in L \) w.l.o.g
- Neighbors of \( s \) must be in \( R \)
- Neighbors of neighbors of \( s \) must be in \( L \)
- \( \ldots \)
- Report “not a bipartite graph” if contradiction was found
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of $s$ must be in $R$
- Neighbors of neighbors of $s$ must be in $L$
- ...  
- Report “not a bipartite graph” if contradiction was found
- If $G$ contains multiple connected components, repeat above algorithm for each component
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness

bad edges!
Testing Bipartiteness using BFS

BFS(s)

1. $\text{head} \leftarrow 1, \text{tail} \leftarrow 1, \text{queue}[1] \leftarrow s$
2. mark $s$ as “visited” and all other vertices as “unvisited”
3. while head $\geq$ tail
4. $v \leftarrow \text{queue}[\text{tail}], \text{tail} \leftarrow \text{tail} + 1$
5. for all neighbours $u$ of $v$
6. if $u$ is “unvisited” then
7. $\text{head} \leftarrow \text{head} + 1, \text{queue}[\text{head}] = u$
8. mark $u$ as “visited”
Testing Bipartiteness using BFS

test-bipartiteness(s)

1. $\text{head} \leftarrow 1, \text{tail} \leftarrow 1, \text{queue}[1] \leftarrow s$
2. mark $s$ as “visited” and all other vertices as “unvisited”
3. $\text{color}[s] \leftarrow 0$
4. while head $\geq$ tail
5. $v \leftarrow \text{queue}[\text{tail}], \text{tail} \leftarrow \text{tail} + 1$
6. for all neighbours $u$ of $v$
7. if $u$ is “unvisited” then
8. $\text{head} \leftarrow \text{head} + 1, \text{queue}[\text{head}] = u$
9. mark $u$ as “visited”
10. $\text{color}[u] \leftarrow 1 - \text{color}[v]$
11. elseif $\text{color}[u] = \text{color}[v]$ then
12. print(“$G$ is not bipartite”) and exit
Testing Bipartiteness using BFS

1. mark all vertices as “unvisited”
2. for each vertex $v \in V$
3.     if $v$ is “unvisited” then
4.         test-bipartiteness($v$)
5.     print(“$G$ is bipartite”)
Testing Bipartiteness using BFS

1. mark all vertices as “unvisited”
2. for each vertex \( v \in V \)
3. if \( v \) is “unvisited” then
4. test-bipartiteness(\( v \))
5. print(“\( G \) is bipartite”)  

**Obs.** Running time of algorithm = \( O(n + m) \)
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Bridges in a Graph
Topological Ordering Problem

**Input:** a directed acyclic graph (DAG) $G = (V, E)$

**Output:** 1-to-1 function $\pi : V \rightarrow \{1, 2, 3 \cdots, n\}$, so that
- if $(u, v) \in E$ then $\pi(u) < \pi(v)$
Topological Ordering Problem

**Input:** a directed acyclic graph (DAG) $G = (V, E)$

**Output:** 1-to-1 function $\pi : V \rightarrow \{1, 2, 3 \cdots, n\}$, so that

- if $(u, v) \in E$ then $\pi(u) < \pi(v)$
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?

A:
- Use linked-lists of outgoing edges
- Maintain the in-degree $d_v$ of vertices
- Maintain a queue (or stack) of vertices $v$ with $d_v = 0$
Let $d_v \leftarrow 0$ for every $v \in V$

for every $v \in V$

    for every $u$ such that $(v, u) \in E$

        $d_u \leftarrow d_u + 1$

    $S \leftarrow \{v : d_v = 0\}$, $i \leftarrow 0$

while $S \neq \emptyset$

    $v \leftarrow$ arbitrary vertex in $S$, $S \leftarrow S \setminus \{v\}$

    $i \leftarrow i + 1$, $\pi(v) \leftarrow i$

    for every $u$ such that $(v, u) \in E$

        $d_u \leftarrow d_u - 1$

    if $d_u = 0$ then add $u$ to $S$

if $i < n$ then output “not a DAG”

$S$ can be represented using a queue or a stack

Running time = $O(n + m)$
### $S$ as a Queue or a Stack

<table>
<thead>
<tr>
<th>DS</th>
<th>Queue</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>$\text{head} \leftarrow 0$, $\text{tail} \leftarrow 1$</td>
<td>$\text{top} \leftarrow 0$</td>
</tr>
<tr>
<td>Non-Empty?</td>
<td>$\text{head} \geq \text{tail}$</td>
<td>$\text{top} &gt; 0$</td>
</tr>
</tbody>
</table>
| Add($v$)    | $\text{head} \leftarrow \text{head} + 1$  
$S[\text{head}] \leftarrow v$ | $\text{top} \leftarrow \text{top} + 1$  
$S[\text{top}] \leftarrow v$ |
| Retrieve $v$ | $v \leftarrow S[\text{tail}]$  
$\text{tail} \leftarrow \text{tail} + 1$ | $v \leftarrow S[\text{top}]$  
$\text{top} \leftarrow \text{top} - 1$ |
Type of edges with respect to a tree

Given a graph $G = (V, E)$ and a rooted tree $T$ in $G$, edges in $G$ can be one of the three types:

- **Tree edges**: edges in $T$
- **Cross edges** $(u, v)$: $u$ and $v$ do not have an ancestor-descendant relation
- **Vertical edges** $(u, v)$: $u$ is an ancestor of $v$, or $v$ is an ancestor of $u$
Type of edges with respect to a tree

Given a graph $G = (V, E)$ and a rooted tree $T$ in $G$, edges in $G$ can be one of the three types:

- **Tree edges**: edges in $T$
- **Cross edges** $(u, v)$: $u$ and $v$ do not have an ancestor-descendant relation
Type of edges with respect to a tree

Given a graph $G = (V, E)$ and a rooted tree $T$ in $G$, edges in $G$ can be one of the three types:

- **Tree edges**: edges in $T$
- **Cross edges** $(u, v)$: $u$ and $v$ do not have an ancestor-descendant relation
- **Vertical edges** $(u, v)$: $u$ is an ancestor of $v$, or $v$ is an ancestor of $u$
Properties of a BFS Tree

Given a tree BFS tree $T$ of a graph $G$,
Properties of a BFS Tree

Given a tree BFS tree $T$ of a graph $G$,

- Can there be vertical edges?

No.

No.

For any cross edge $(u,v)$, $u$ and $v$ are at most 1 level apart.
Properties of a BFS Tree

Given a tree BFS tree $T$ of a graph $G$,

- Can there be vertical edges?
  - No.

For any cross edge $(u,v)$, $u$ and $v$ are at most 1 level apart.
Properties of a BFS Tree

Given a tree BFS tree $T$ of a graph $G$,

- Can there be vertical edges?
  - No.
Properties of a BFS Tree

Given a tree BFS tree $T$ of a graph $G$,

- Can there be vertical edges?
  - No.
- Can there be cross edges $(u, v)$ with $u$ and $v$ 2 levels apart?

![BFS Tree Diagram](image-url)
Properties of a BFS Tree

Given a tree BFS tree $T$ of a graph $G$,

- Can there be vertical edges?
  - No.
- Can there be cross edges $(u, v)$ with $u$ and $v$ 2 levels apart?
  - No.
Properties of a BFS Tree

Given a tree BFS tree $T$ of a graph $G$,

- Can there be vertical edges?  
  No.

- Can there be cross edges $(u, v)$ with $u$ and $v$ 2 levels apart?  
  No.
Properties of a BFS Tree

Given a tree BFS tree $T$ of a graph $G$,

- Can there be vertical edges?
  - No.

- Can there be cross edges $(u, v)$ with $u$ and $v$ 2 levels apart?
  - No.

- For any cross edge $(u, v)$, $u$ and $v$ are at most 1 level apart.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph $G$,
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph $G$,

- Can there be cross edges?
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph $G$,
- Can there be cross edges?
- No.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph $G$,

- Can there be cross edges?
- No.
Given a tree DFS tree $T$ of a graph $G$,

- Can there be cross edges?
  - No.

- All non-tree edges are vertical edges.
Def. Given a connected graph \( G = (V, E) \), an edge \( e \in E \) is called a bridge if the graph \( G = (V, E \setminus \{e\}) \) is disconnected.
Def. Given a connected graph $G = (V, E)$, an edge $e \in E$ is called a bridge if the graph $G = (V, E \setminus \{e\})$ is disconnected.
There are only tree edges and vertical edges.
- There are only tree edges and vertical edges
- Vertical edges are not bridges
There are only tree edges and vertical edges.

Vertical edges are not bridges.

A tree edge \((v, u)\) is not a bridge if some vertical edge jumping from below \(u\) to above \(v\) is not a bridge.
There are only tree edges and vertical edges.
Vertical edges are not bridges.
A tree edge \((v, u)\) is not a bridge if some vertical edge jumping from below \(u\) to above \(v\) is not a bridge.
Other tree edges are bridges.
level (v): the level of vertex v in DFS tree
T_v: the subtree rooted at v
h(v): the smallest level that can be reached using a vertical edge from vertices in T_v (parent, u) is a bridge if h(u) ≥ level(u).
- $level(v)$: the level of vertex $v$ in DFS tree
- \( \text{level}(v) \): the level of vertex \( v \) in DFS tree

- \( T_v \): the sub tree rooted at \( v \)

- \( h(v) \): the smallest level that can be reached using a vertical edge from vertices in \( T_v \)
- **$level(v)$**: the level of vertex $v$ in DFS tree
- **$T_v$**: the sub tree rooted at $v$
- **$h(v)$**: the smallest level that can be reached using a vertical edge from vertices in $T_v$
- **$(parent(u), u)$** is a bridge if $h(u) \geq level(u)$. 
recursive-DFS($v$)

1. mark $v$ as “visited”
2. $h(v) \leftarrow \infty$
3. for all neighbours $u$ of $v$
   4. if $u$ is unvisited then
      5. $level(u) \leftarrow level(v) + 1$
      6. recursive-DFS($u$)
      7. if $h(u) \geq level(u)$ then claim $(v, u)$ is a bridge
     if $h(u) < h(v)$ then $h(v) \leftarrow h(u)$
   9. else if $level(u) < level(v) - 1$ then
      10. if $level(u) < h(v)$ then $h(v) \leftarrow level(u)$
Finding Bridges

1. mark all vertices as “unvisited”
2. for every $v \in V$ do
3.   if $v$ is unvisited then
4.     $\text{level}(v) \leftarrow 0$
5.     recursive-DFS($v$)