CSE 431/531: Algorithm Analysis and Design (Spring 2021)

Graph Basics

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo
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
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 $v$. 

---

1: \[2 \rightarrow 3\]
2: \[1 \rightarrow 3 \rightarrow 4 \rightarrow 5\]
3: \[1 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 8\]
4: \[2 \rightarrow 5\]
5: \[2 \rightarrow 3 \rightarrow 4 \rightarrow 6\]
6: \[5\]
7: \[3 \rightarrow 8\]
8: \[3 \rightarrow 7\]
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>( 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>( 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 neighbors 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
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)

**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 $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- $L_{j+1}$ 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 $L_0, L_1, L_2, L_3, \ldots$
- $L_0 = \{s\}$
- $L_{j+1}$ 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 $L_j$
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 $head \geq tail$ do
4: $v \leftarrow queue[tail], tail \leftarrow tail + 1$
5: for all neighbours $u$ of $v$ do
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

![Graph Diagram]

- **V**: Vertex
- **Head**: Queue Head
- **Tail**: Queue Tail
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

\[ \text{head} \]

\[ \text{tail} \]
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

The diagram on the left shows a graph with nodes labeled 1 to 8. The node labeled 7 is highlighted. The sequence of nodes visited during a breadth-first search (BFS) is shown on the right. The sequence is: 1, 2, 3, 4, 5, 7, 8, 6.
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
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

![Graph Diagram]
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 Recurrsion

**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 do
3: if u is unvisited then recursive-DFS(u)
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering
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 \( s \in V \)
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 \( R \)

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 \( L \)
- \( \ldots \)
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: head ← 1, tail ← 1, queue\([1]\) ← s
2: mark s as “visited” and all other vertices as “unvisited”
3: while head ≥ tail do
4: \(v ← queue[tail], tail ← tail + 1\)
5: for all neighbours \(u\) of \(v\) do
6: \(\text{if } u \text{ is “unvisited” then}\)
7: \(head ← head + 1, queue[head] = u\)
8: mark \(u\) as “visited”
Testing Bipartiteness using BFS

test-bipartiteness(s)

1: head ← 1, tail ← 1, queue[1] ← s
2: mark s as “visited” and all other vertices as “unvisited”
3: color[s] ← 0
4: while head ≥ tail do
5:   v ← queue[tail], tail ← tail + 1
6:   for all neighbours u of v do
7:     if u is “unvisited” then
8:       head ← head + 1, queue[head] = u
9:       mark u as “visited”
10:    color[u] ← 1 − color[v]
11:   else if color[u] = 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 \( v \in V \) do
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 \( v \in V \) do
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
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.
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.
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.
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.

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 $d_v = 0$. 
Topological Ordering

- 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 $d_v = 0$. 
Topological Ordering

- 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 $d_v = 0$
topological-sort($G$)

1: let $d_v \leftarrow 0$ for every $v \in V$
2: for every $v \in V$ do
3: for every $u$ such that $(v, u) \in E$ do
4: $d_u \leftarrow d_u + 1$
5: $S \leftarrow \{v : d_v = 0\}$, $i \leftarrow 0$
6: while $S \neq \emptyset$ do
7: $v \leftarrow$ arbitrary vertex in $S$, $S \leftarrow S \setminus \{v\}$
8: $i \leftarrow i + 1$, $\pi(v) \leftarrow i$
9: for every $u$ such that $(v, u) \in E$ do
10: $d_u \leftarrow d_u - 1$
11: if $d_u = 0$ then add $u$ to $S$
12: 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>$head \leftarrow 0, \ tail \leftarrow 1$</td>
<td>$top \leftarrow 0$</td>
</tr>
<tr>
<td>Non-Empty?</td>
<td>$head \geq tail$</td>
<td>$top &gt; 0$</td>
</tr>
<tr>
<td>Add($v$)</td>
<td>$head \leftarrow head + 1$</td>
<td>$top \leftarrow top + 1$</td>
</tr>
<tr>
<td></td>
<td>$S[head] \leftarrow v$</td>
<td>$S[top] \leftarrow v$</td>
</tr>
<tr>
<td>Retrieve $v$</td>
<td>$v \leftarrow S[tail]$</td>
<td>$v \leftarrow S[top]$</td>
</tr>
<tr>
<td></td>
<td>$tail \leftarrow tail + 1$</td>
<td>$top \leftarrow top - 1$</td>
</tr>
</tbody>
</table>
Example

\[
\begin{array}{c}
\text{queue:} \\
| a | b | c | d | e | f | g \\
\text{degree} | 0 | 1 | 1 | 1 | 2 | 1 | 3 \\
\end{array}
\]
Example

```latex
\begin{center}
\begin{tikzpicture}[->,>=stealth,auto]
\node[shape=circle,draw=black] (a) {$a$};
\node[shape=circle,draw=black] (b) [below left of=a] {$b$};
\node[shape=circle,draw=black] (c) [below right of=a] {$c$};
\node[shape=circle,draw=black] (d) [below left of=c] {$d$};
\node[shape=circle,draw=black] (e) [below right of=d] {$e$};
\node[shape=circle,draw=black] (f) [below right of=a] {$f$};
\node[shape=circle,draw=black] (g) [below right of=f] {$g$};
\draw (a) to (b);
\draw (c) to (b);
\draw (c) to (a);
\draw (c) to (f);
\draw (d) to (c);
\draw (d) to (e);
\draw (d) to (f);
\draw (e) to (d);
\draw (e) to (f);
\end{tikzpicture}
\end{center}

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
& $a$ & $b$ & $c$ & $d$ & $e$ & $f$ & $g$ \\
\hline
degree & 0 & 1 & 1 & 1 & 2 & 1 & 3 \\
\hline
\end{tabular}
\end{center}
```
Example

A graph with nodes labeled $b$, $c$, $d$, $f$, and $g$ is shown. The edges connect nodes as follows: $b$ to $d$, $d$ to $c$, $c$ to $d$, $f$ to $g$, and $g$ to $e$. The queue is represented as $a$ with head and tail indicators.

The degree of each node is listed in the table:

<table>
<thead>
<tr>
<th>degree</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
<th>$g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

The graph and queue demonstrate concepts such as adjacency and queue operations in a network.
Example

```
degree
0  0  0  1  2  1  3

queue:
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

head

tail
```
Example

![Graph Diagram]

- **Queue:**
  - `a`, `b`, `c`

- **Degree Table:**
  - | `a` | `b` | `c` | `d` | `e` | `f` | `g` |
  - | 0   | 0   | 0   | 1   | 2   | 1   | 3   |
Example
Example

![Graph Diagram]

- **Queue:**
  - `head`: `a`
  - `tail`: `g`

- **Degree Table:**
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>degree</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

```
d e g
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
```

**queue:**
```
head
```

**degree**
```
tail
```
Example
Example
Example

- Queue:
  - head
  - tail
  - Elements: a, b, c, d, f

- Degree:
  - Values: 0, 0, 0, 0, 0, 0, 0, 2
Example
Example

queue: \[
a & b & c & d & f & e
\]

degree
\[
\begin{array}{cccccccc}
  a & b & c & d & e & f & g \\
  0 & 0 & 0 & 0 & 0 & 0 & 2 \\
\end{array}
\]
Example

queue: | a | b | c | d | f | e |

degree | 0 | 0 | 0 | 0 | 0 | 0 | 1

head

tail
Example

queue: [a b c d f e]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

degree

e
f
g
head
tail

\( e \rightarrow g \)
Example

Queue:

\[
\begin{array}{ccccccc}
    & a & b & c & d & f & e \\
\end{array}
\]

\[\text{degree} \quad \begin{array}{ccccccc}
    0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}\]
**Example**

**Queue:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f</th>
<th>e</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Head</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tail</strong></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
</tr>
</tbody>
</table>

**Degree:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Degree</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

queue: \[ a \ b \ c \ d \ f \ e \ g \]

\( g \)

\[
\begin{array}{l}
\text{degree} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]