Graph ADT

**Data:**
- Vertices
- Edges
- Some data structure maintaining the structure between vertices and edges.

**Functions:**
- `insertVertex(K key);`
- `insertEdge(Vertex v1, Vertex v2, K key);`
- `removeVertex(Vertex v);`
- `removeEdge(Vertex v1, Vertex v2);`
- `incidentEdges(Vertex v);`
- `areAdjacent(Vertex v1, Vertex v2);`
- `origin(Edge e);`
- `destination(Edge e);`
**Key Ideas:**
- Given a vertex, $O(1)$ lookup in vertex list
  - Implement with a hash table, etc
- All basic ADT operations run in $O(m)$ time
Key Ideas:
- Given a vertex, $O(1)$ lookup in vertex list
- Given a pair of vertices (an edge), $O(1)$ lookup in the matrix
- Undirected graphs can use an upper triangular matrix
Graph Implementation: Edge List

![Graph Diagram]

- Vertices: V, W, Z
- Edges: (u, v), (v, w), (w, z), (u, c), (w, d)

Edge List Representation:

<table>
<thead>
<tr>
<th>u</th>
<th>v</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>w</td>
<td>b</td>
</tr>
<tr>
<td>u</td>
<td>w</td>
<td>c</td>
</tr>
<tr>
<td>w</td>
<td>z</td>
<td>d</td>
</tr>
</tbody>
</table>
Adjacency List

```
<table>
<thead>
<tr>
<th></th>
<th>u</th>
<th>v</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>v</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>u</td>
<td>w</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>v</td>
<td>z</td>
<td>d</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>u</th>
<th>v</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>v</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>u</td>
<td>w</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>v</td>
<td>z</td>
<td>d</td>
</tr>
</tbody>
</table>
```
Adjacency List

The diagram illustrates an adjacency list representation of a graph. Each node in the graph is represented by a square, and the edges are shown with arrows. The adjacency list is shown in the box on the right, listing the neighbors of each node:

- Node u has neighbors v, w, and c.
- Node v has neighbors u and w.
- Node w has neighbors u, v, and c.
- Node z has neighbors w and d.
- Node d has no explicit neighbors in this diagram.

The values 'd=1', 'd=2', and 'd=3' indicate the degree of the nodes in the graph.
Adjacency List

insertVertex(K key):

- u
- v
- w
- z
Adjacency List

removeVertex(Vertex v):

Vertex u
- a (d=2)
- c (d=2)

Vertex v
- a (d=2)
- b (d=2)

Vertex w
- b (d=3)
- c
- d (d=3)

Vertex z
- d (d=1)

Vertex a
- c

Vertex b
- c

Vertex c
- d

Vertex d
- z
- w
Adjacency List

incidentEdges(Vertex v):
Adjacency List

areAdjacent(Vertex v1, Vertex v2):
insertEdge(Vertex v1, Vertex v2, K key):
<table>
<thead>
<tr>
<th>Operation</th>
<th>Edge List</th>
<th>Adjacency Matrix</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>n+m</td>
<td>n^2</td>
<td>n+m</td>
</tr>
<tr>
<td>insertVertex(v)</td>
<td>1</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>removeVertex(v)</td>
<td>m</td>
<td>n</td>
<td>deg(v)</td>
</tr>
<tr>
<td>insertEdge(v, w, k)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>removeEdge(v, w)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>incidentEdges(v)</td>
<td>m</td>
<td>n</td>
<td>deg(v)</td>
</tr>
<tr>
<td>areAdjacent(v, w)</td>
<td>m</td>
<td>1</td>
<td>min( deg(v), deg(w) )</td>
</tr>
</tbody>
</table>
Traversals:

**Objective:** Visit every vertex and every edge in the graph.

**Purpose:** Search for interesting sub-structures in the graph.

We’ve seen traversal before ....but it’s different:

- Ordered
- Obvious Start

- •
- •
- •
Traversal: BFS

Diagram of graph with nodes A, B, C, D, E, F, G, H and edges connecting them.
Traversals: BFS

<table>
<thead>
<tr>
<th>v</th>
<th>d</th>
<th>P</th>
<th>Adjacent Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Traversals: BFS

<table>
<thead>
<tr>
<th>d</th>
<th>p</th>
<th>Adjacent Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>A CBD</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>B A CE</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>C BA DEF</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>D ACFH</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>E BCG</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>F CDG</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>G EFH</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>H DG</td>
</tr>
</tbody>
</table>
BFS(G):
  Input: Graph, G  
  Output: A labeling of the edges on G as discovery and cross edges

  foreach (Vertex v : G.vertices()):
      setLabel(v, UNEXPLORED)
  foreach (Edge e : G.edges()):
      setLabel(e, UNEXPLORED)
  foreach (Vertex v : G.vertices()):
      if getLabel(v) == UNEXPLORED:
          BFS(G, v)

BFS(G, v):
  Queue q
  setLabel(v, VISITED)
  q.enqueue(v)

  while !q.empty():
      v = q.dequeue()
      foreach (Vertex w : G.adjacent(v)):
          if getLabel(w) == UNEXPLORED:
              setLabel(v, w, DISCOVERY)
              setLabel(w, VISITED)
              q.enqueue(w)
          elseif getLabel(v, w) == UNEXPLORED:
              setLabel(v, w, CROSS)
BFS Analysis

Q: Does our implementation handle disjoint graphs? If so, what code handles this?
   • *How do we use this to count components?*

Q: Does our implementation detect a cycle?
   • *How do we update our code to detect a cycle?*

Q: What is the running time?
Running time of BFS

While-loop at :19?

For-loop at :21?
BFS(G):

Input: Graph, G
Output: A labeling of the edges on G as discovery and cross edges

foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
foreach (Vertex v : G.vertices()):
    if.getLabel(v) == UNEXPLORED:
        BFS(G, v)

BFS(G, v):

Queue q
setLabel(v, VISITED)
q.enqueue(v)

while !q.empty():
    v = q.dequeue()
    foreach (Vertex w : G.adjacent(v)):
        if getLabel(w) == UNEXPLORED:
            setLabel(v, w, DISCOVERY)
            setLabel(w, VISITED)
            q.enqueue(w)
        elseif getLabel(v, w) == UNEXPLORED:
            setLabel(v, w, CROSS)
BFS Observations

Q: What is a shortest path from A to H?

Q: What is a shortest path from E to H?

Q: How does a cross edge relate to d?

Q: What structure is made from discovery edges?
BFS Observations

**Obs. 1:** Traversals can be used to count components.

**Obs. 2:** Traversals can be used to detect cycles.

**Obs. 3:** In BFS, $d$ provides the shortest distance to every vertex.

**Obs. 4:** In BFS, the endpoints of a cross edge never differ in distance, $d$, by more than 1:

$$|d(u) - d(v)| = 1$$
Traversal: DFS
BFS(G):
Input: Graph, G
Output: A labeling of the edges on G as discovery and cross edges

foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
foreach (Vertex v : G.vertices()):
    if getLabel(v) == UNEXPLORED:
        BFS(G, v)

BFS(G, v):
Queue q
setLabel(v, VISITED)
q.enqueue(v)

while !q.empty():
    v = q.dequeue()
    foreach (Vertex w : G.adjacent(v)):
        if getLabel(w) == UNEXPLORED:
            setLabel(v, w, DISCOVERY)
            setLabel(w, VISITED)
            q.enqueue(w)
        elseif getLabel(v, w) == UNEXPLORED:
            setLabel(v, w, CROSS)
DFS(G):
Input: Graph, G
Output: A labeling of the edges on G as discovery and back edges

foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
foreach (Vertex v : G.vertices()):
    if getLabel(v) == UNEXPLORED:
        DFS(G, v)

DFS(G, v):
Queue q
setLabel(v, VISITED)
q.enqueue(v)
while !q.empty():
    v = q.dequeue()
    foreach (Vertex w : G.adjacent(v)):
        if getLabel(w) == UNEXPLORED:
            setLabel(v, w, DISCOVERY)
            setLabel(w, VISITED)
            DFS(G, w)
        elseif getLabel(v, w) == UNEXPLORED:
            setLabel(v, w, BACK)
Running time of DFS

Labeling:
• Vertex:
• Edge:

Queries:
• Vertex:
• Edge: