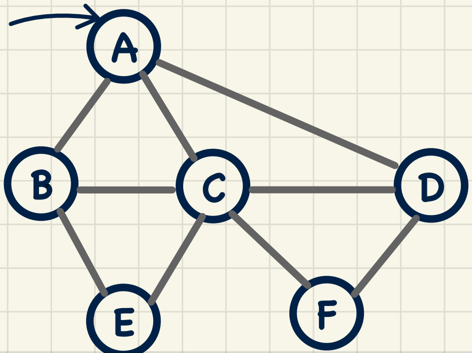
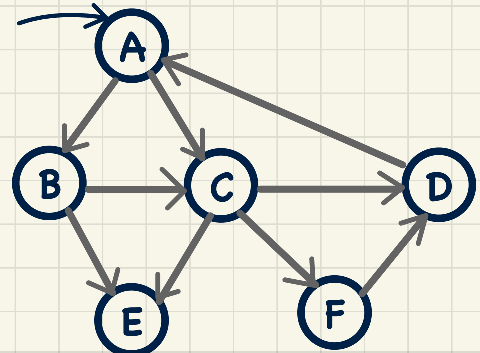
# Back Edge (DFS) vs. Cross Edge (BFS): Cyclic?

Does a back edge always imply the existence of a cycle?

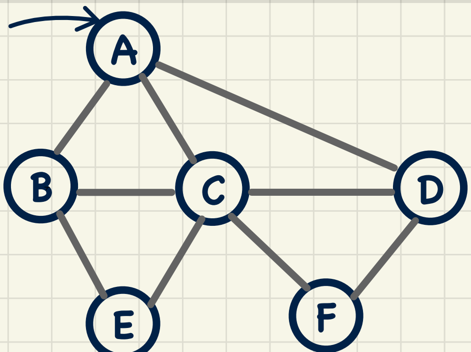
Does a cross edge always imply the existence of a cycle?

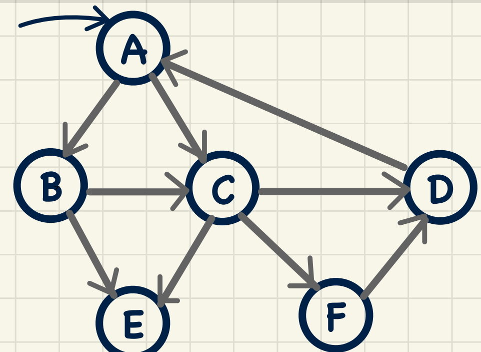
## DFS




## BFS



# Graphs in Java: Adjacency List Strategy (1)

```
class AdjacencyListGraph<V, E> implements Graph<V, E> {
  private DoublyLinkedList<AdjacencyListVertex<V>> vertices;
  private DoublyLinkedList<AdjacencyListEdge<E, V>> edges;
  private boolean isDirected;

  /* initialize an empty graph */
  AdjacencyListGraph(boolean isDirected) {
    vertices = new DoublyLinkedList<>();
    edges = new DoublyLinkedList<>();
    this.isDirected = isDirected;
  }
```

```
public class Vertex<V> {
  private V element;
  public Vertex(V element) { this.element = element; }
  /* setter and getter for element */
}
```

```
public class EdgeListVertex<V> extends Vertex<V> {
  public DLNode<Vertex<V>> vertextListPosition;
  /* setter and getter for vertexListPosition */
}
```

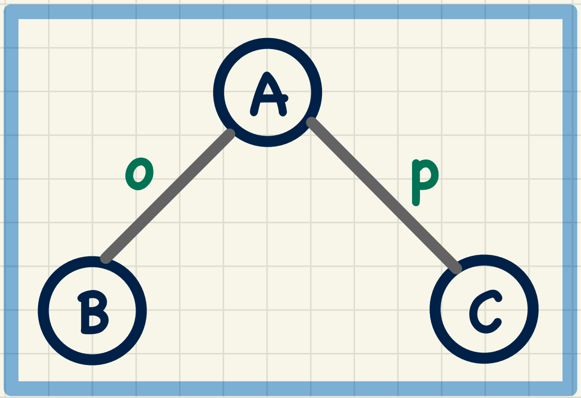
```
class AdjacencyListVertex<V> extends EdgeListVertex<V> {
  private DoublyLinkedList<AdjacencyListEdge<E, V>> incidentEdges;
  /* getter for incidentEdges */
}
```

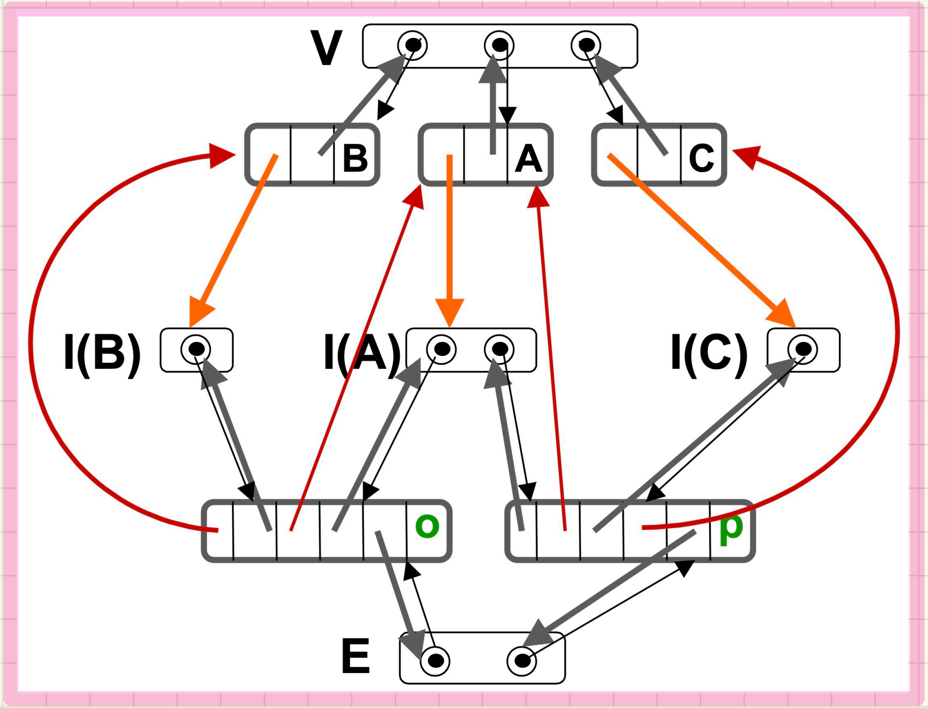
```
public class Edge<E, V> {
  private E element;
  private Vertex<V> origin;
  private Vertex<V> dest;
  public Edge(E element) { this.element = element; }
  /* setters and getters for element, origin, and destination */
}
```

```
public class EdgeListEdge<E, V> extends Edge<E, V> {
  public DLNode<Edge<E, V>> edgeListPosition;
  /* setter and getter for edgeListPosition */
}
```

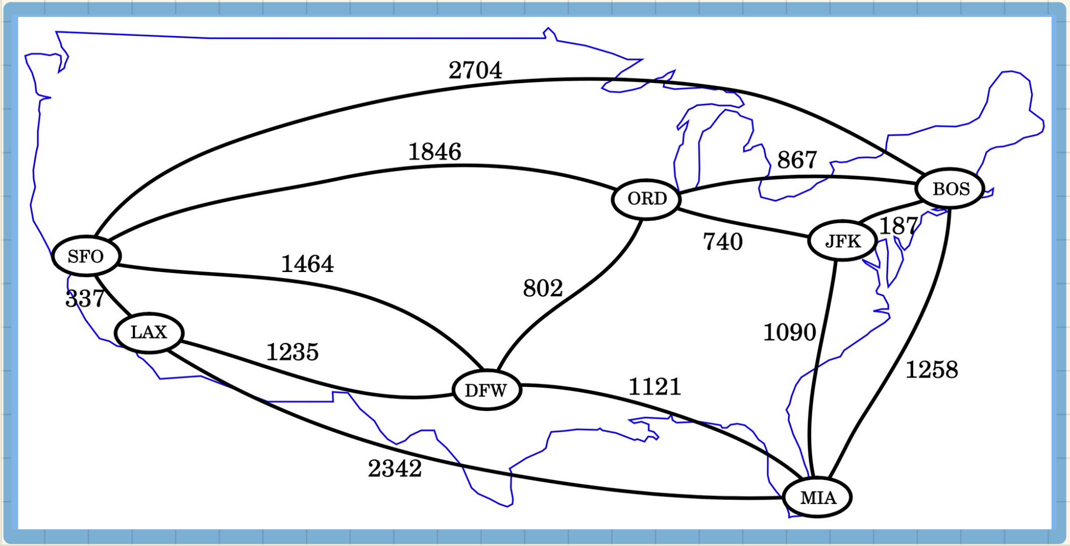
```
class AdjacencyListEdge<V> extends EdgeListEdge<V> {
  DLNode<Edge<E, V>> originIncidentListPos;
  DLNode<Edge<E, V>> destIncidentListPos;
}
```

# Graphs in Java: Adjacency List Strategy (2)



# Shortest Paths in Weighted Graphs



# Dijkstra's Shortest Path Algorithm

Starting from a ***source vertex s***, perform a ***BFS***-like procedure:

**1.** Initially:

**1.1** Set $D(s) = 0$, and every other vertex $t \neq s$, $D(t) = \infty$. [***distance***]

**1.2** Set $a(v) = nil$ for every vertex $v$. [***ancestor*** in *shortest path*]

**1.3** Insert all vertices into a ***priority queue*** $Q$ [ ***key***ed by $D$ ]

**2.** While $Q$ is not empty, repeat the following:

**2.1** Find vertex $u$ in $Q$ s.t. $D(u)$ is the **minimum**.

**2.2** For every vertex $v$ ***adjacent to*** $u$, **if**:

$v \notin Q \wedge \boxed{D(u) + w(u, v) < D(v)}$, then:

- Set $D(v) = D(u) + w(u, v)$
- Set $a(v) = u$

**2.3** Remove vertex $u$ from $Q$.

Upon completion, for every vertex $t$ ($t \neq s$):

- $\boxed{D(t) = d(s, t)}$ (i.e., weight of *shortest path* from $s$ to $t$).
- **Reversing** $t$'s ***ancestor path*** → *shortest path*: $\langle s, \ldots, a(t), t \rangle$

# Dijkstra's Shortest Path Algorithm: Example 1