# Lecture 14 - Nov 3

## *Graphs*

***Tracing BFS using a FIFO Queue***
***Back Edge (DFS) vs. Cross Edge (BFS)***
***Implementing Graphs: Adjacency Lists***

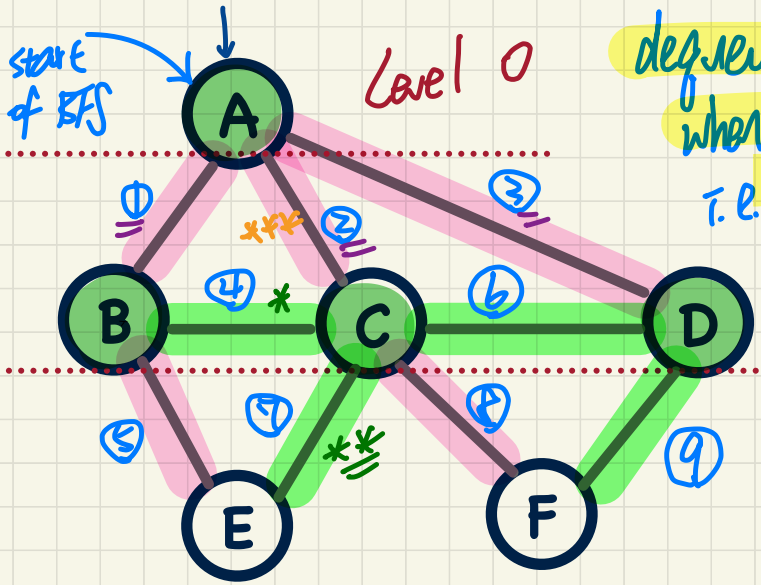
## Announcements/Reminders

- Today's class: notes template posted
- Test 1 results to be released on Tuesday (Nov 4)
- Change of Dates:
  - + Assignment 2 to be released on Wed, Nov 12
  - + Assignment 2 to be due on Wed, Nov 19
  - + Test 2 to be take place on Mon, Nov 24

*** not possible to have a cross edge linking back to a previous level

* cross edge connecting vertices at the same level.

# Breadth-First Search (BFS): Example 1 (a)




dequeue a vertex when all its adj. i.e. have been marked.

| enqueued | dequeued |
|---|---|
| A | A |
| B | B |
| C | C |
| D | D |
| E | E |
| F | F |

** cross edge connecting vertices at the next level.

Assumptions:

- Adjacent vertices visited in alphabetic order

front

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| | ① | ② | ③ | ⑤ | ⑧ | |

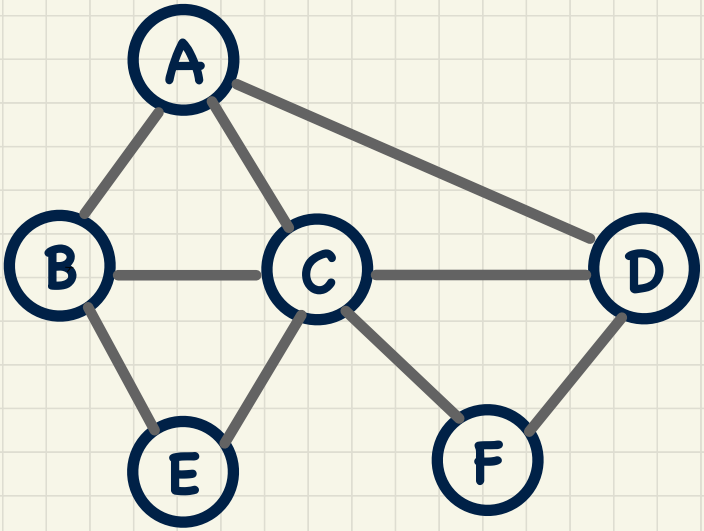
L0 L1 L2

vertices 1 edge away from A

vertices 2 edges away from A.

back

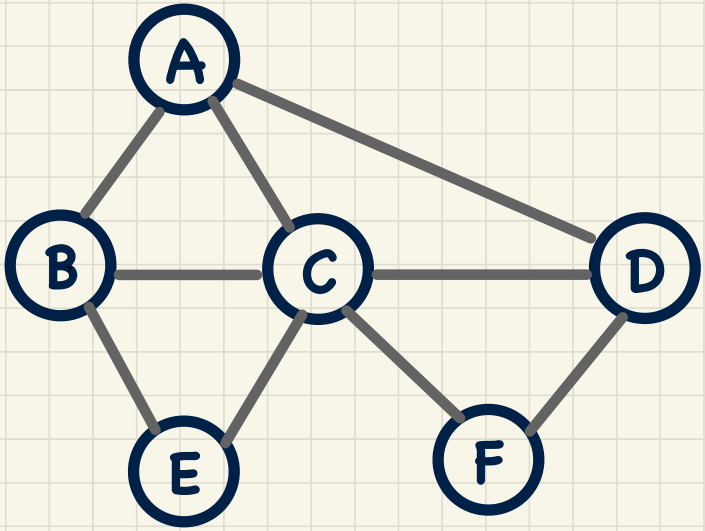
# Breadth-First Search (BFS): Example 1 (b)




Assumptions:

- Adjacent vertices visited in alphabetic order
- Exception: Edge AC visited first

# Breadth-First Search (BFS): Example 1 (c)

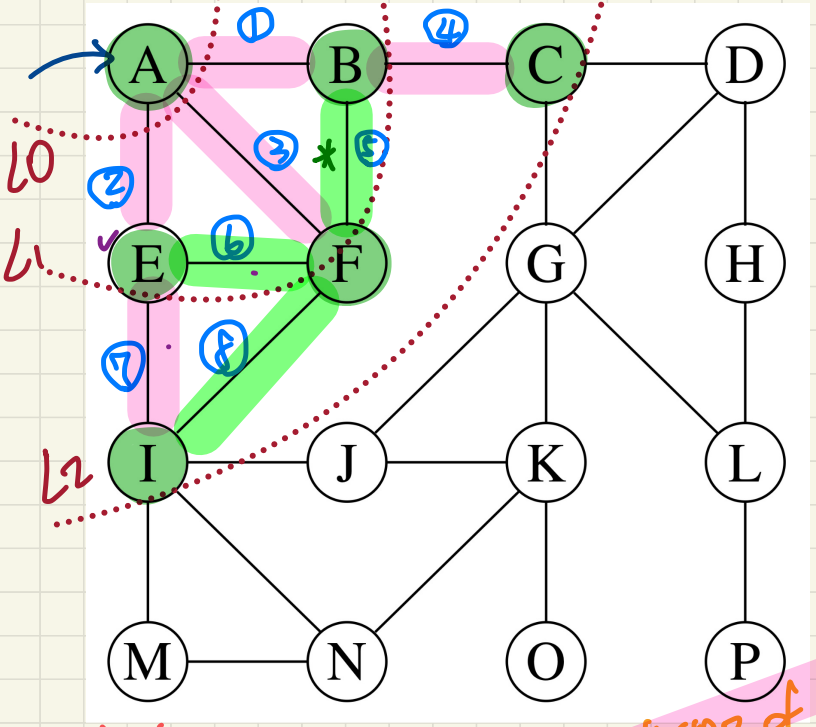



Assumptions:

- Adjacent vertices visited in alphabetic order
- Exception: Edge AD visited first

# Breadth-First Search (BFS): Example 2

* Though organized vertically, vertices B and F are at the same level.




| enqueued | dequeued |
|---|---|
| A | A |
| B ① | B |
| E ② | E |
| F ③ | F |
| C ④ | |

front

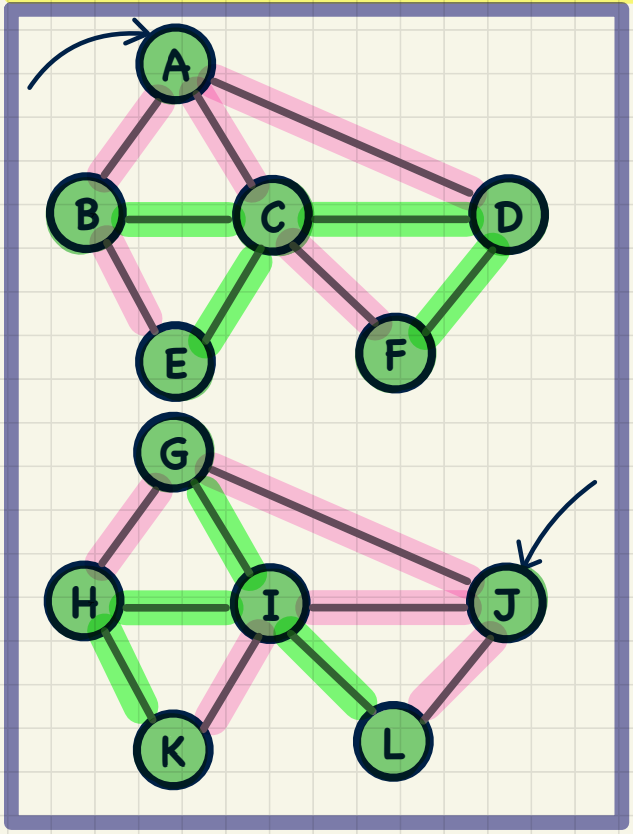
① ② ③ ④ ⑦

~~A~~ ~~B~~ ~~E~~ ~~F~~ C 1

next step of BFS: go over each vertex at L2 and reach out to L3

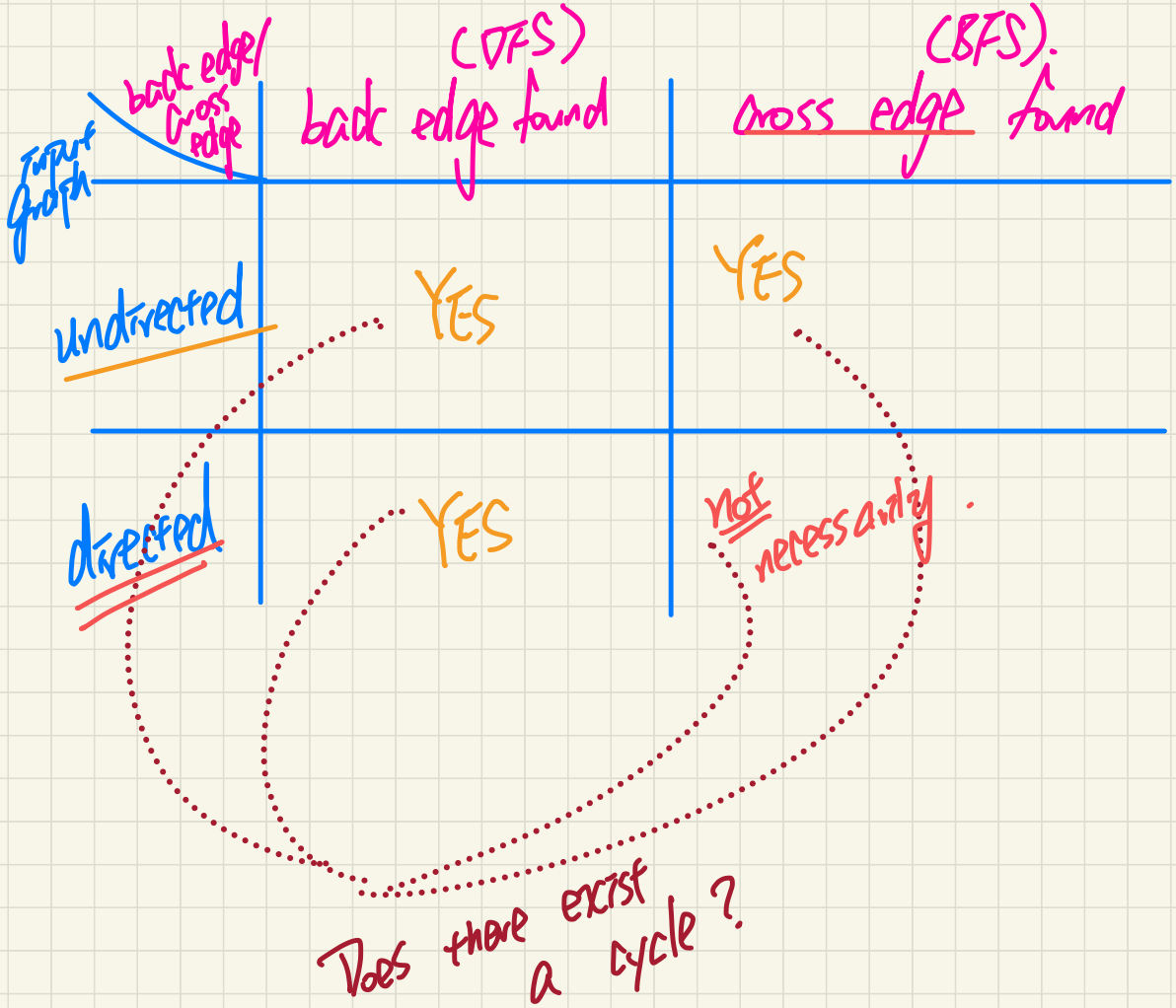
# Graph Traversals: Adapting BFS

## Efficient Traversal of Graph G:




### Graph Questions:

- Fina a path between {u, v} ⊆ V
  Start BFS from u, maintain the path until v is encountered.
- Is v reachable from v
  Start BFS from u, is v ever encountered?
- Find all connected components of G.
  Keep doing BFS, until all vertices are visited.
- Compute a spanning tree of a connected G.
  Each BFS will identify the spanning tree containing the start vertex
- Is G connected?
  How many BFSes are needed to cover all vertices? (1)
- If G is cyclic, return a cycle.


back edge/
cross edge
(DFS)
back edge found
(BFS).
cross edge found
Input graph
undirected
directed
YES
YES
YES
not necessarily
Does there exist a cycle?

```
int numVertexVisited = 0;
while (numVertexVisited < |V|) {
    Pick some vertex x (unvisited),
        start BFS on x.
}
```

exit loop as soon as numV. Visited = |V|

↓ numVertexVisited properly updated (for every discovery edge)

# iterations = # CCs.

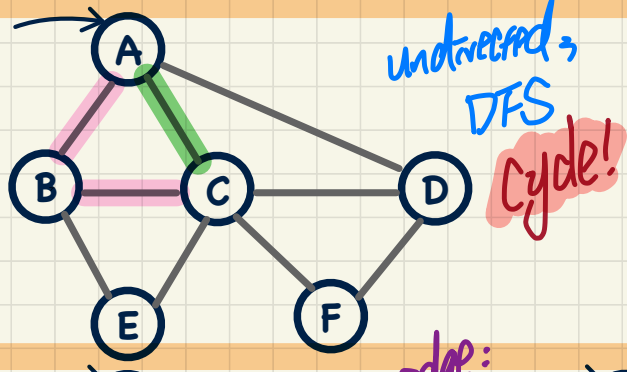
# Back Edge (DFS) vs. Cross Edge (BFS): Cyclic?

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

YES.

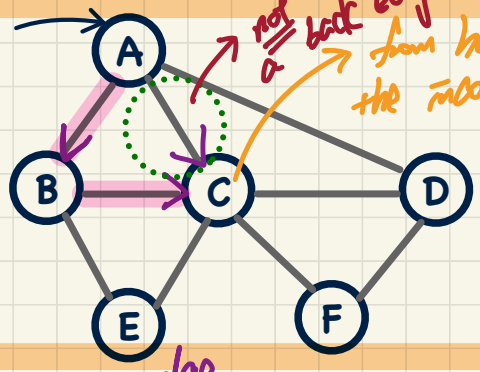
focus on finding a path leading to a back edge.

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



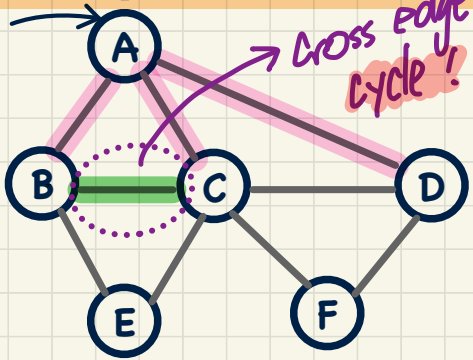

undirected → DFS

Cycle!

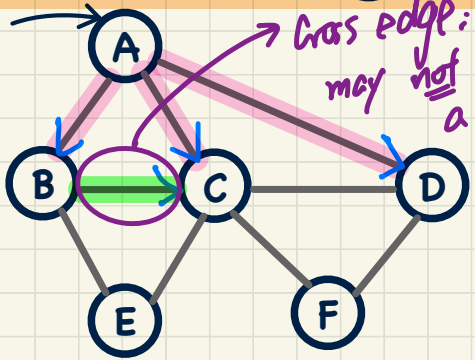



not a back edge.

from here the incoming edge will not be considered as a back edge.




Cross edge: cycle!




Cross edge: may not be a cycle

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

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

for efficient removal of edges.

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

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