# Data Structures & Algorithms

Adil M. Khan

Professor of Computer Science

Innopolis University
a.khan@innopolis.ru

# Recap

- Graphs and related terminologies/properties
- Graphs as an ADT
- Graphs Representations
  - Edge List
  - Adjacency List
  - Adjacency Matrix

# Graph Traversals

# Objectives

1. Why do we need to “traverse” a graph

2. Learn and analyze two ways to traverse a graph

- Depth First Search (DFS)

- Breadth First Search (BFS)

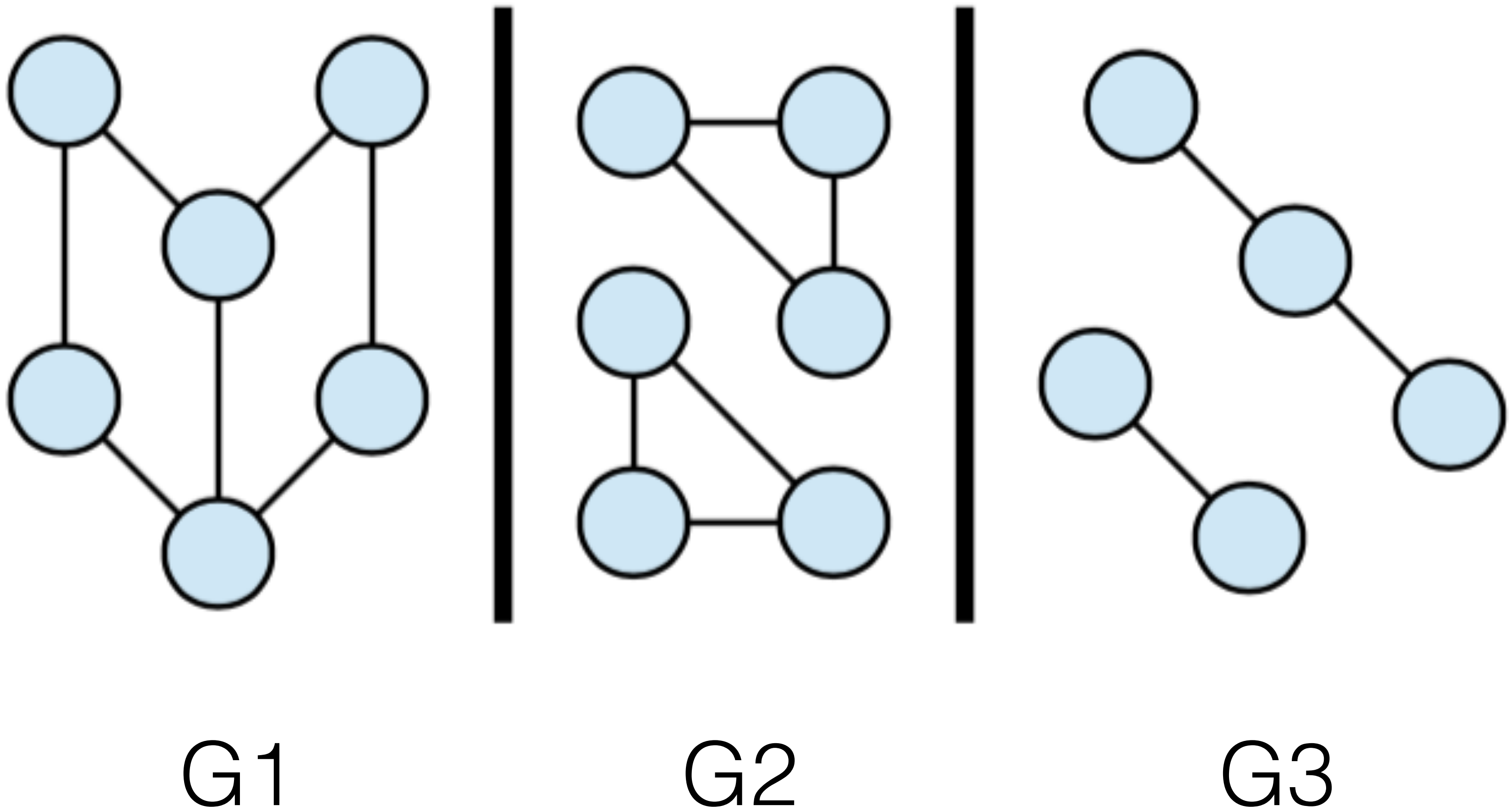
# Why “Traverse”

- Graph node **traversal** is not important by itself. For any implementation, it is possible just to iterate through the list of nodes. Usually, traversal is a way to answer the questions related to graph connectivity

  - Is this a **connected** graph? (Can I visit all nodes/exact node from here?)
  - Find the **path** from here to everywhere/to destination
  - Show me my **nearest context** (e.g. 2 hops around)
  - Are there **cycles** in a graph? (Circular references)
  - Are there **bridges**? (Threat detection)

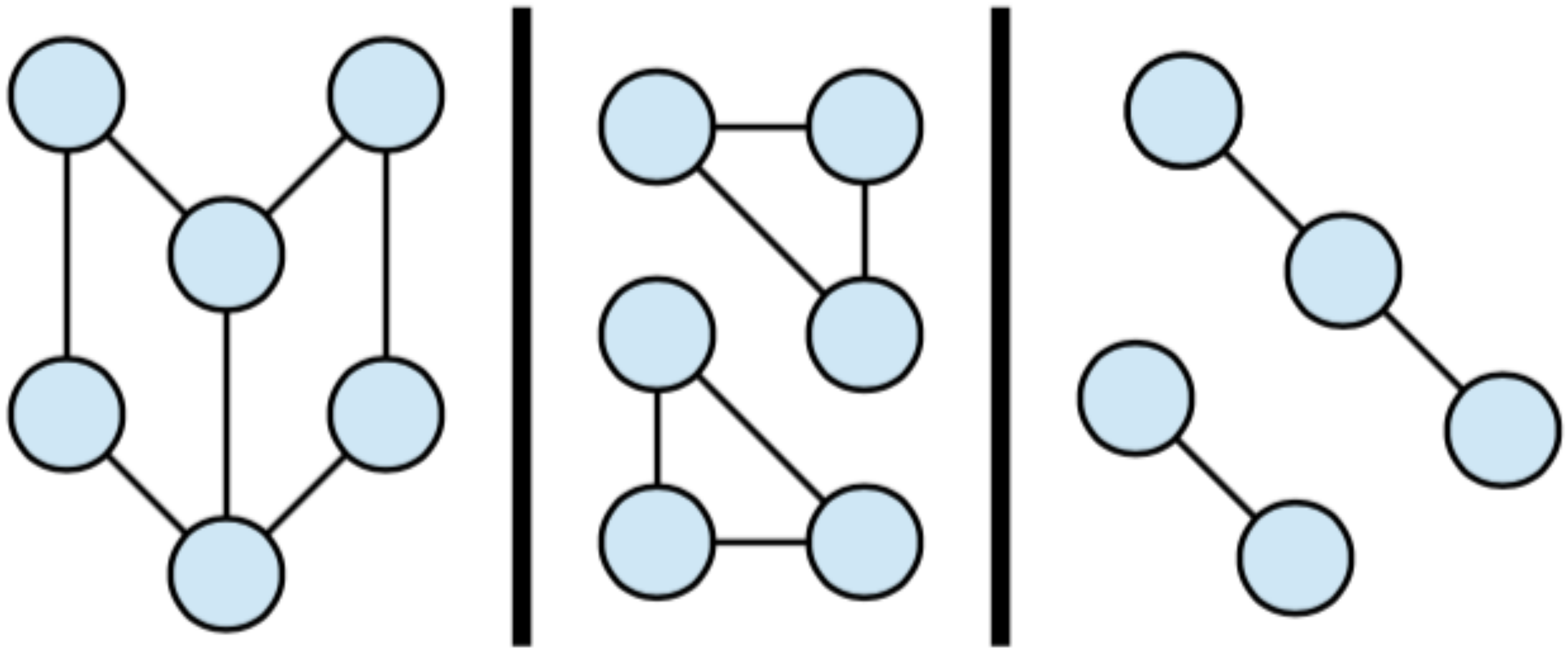
# Connected Component

- Let G be an undirected graph.
- Two nodes u and v are called **connected** if there is a path from u to v in G (u $\longleftrightarrow$ v)
- Now consider the following graphs



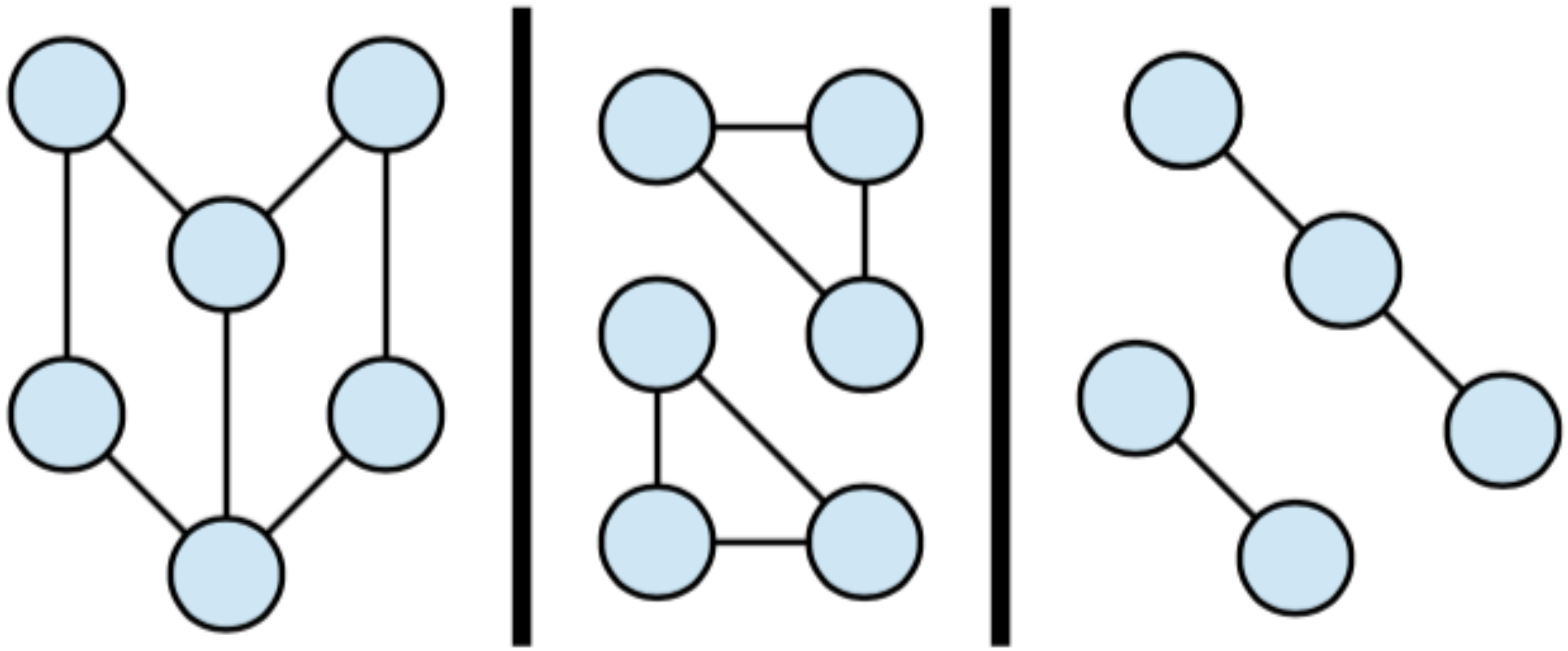
G1 seems like it is one big piece.

G2 and G3 are in multiple pieces.



G1 G2 G3

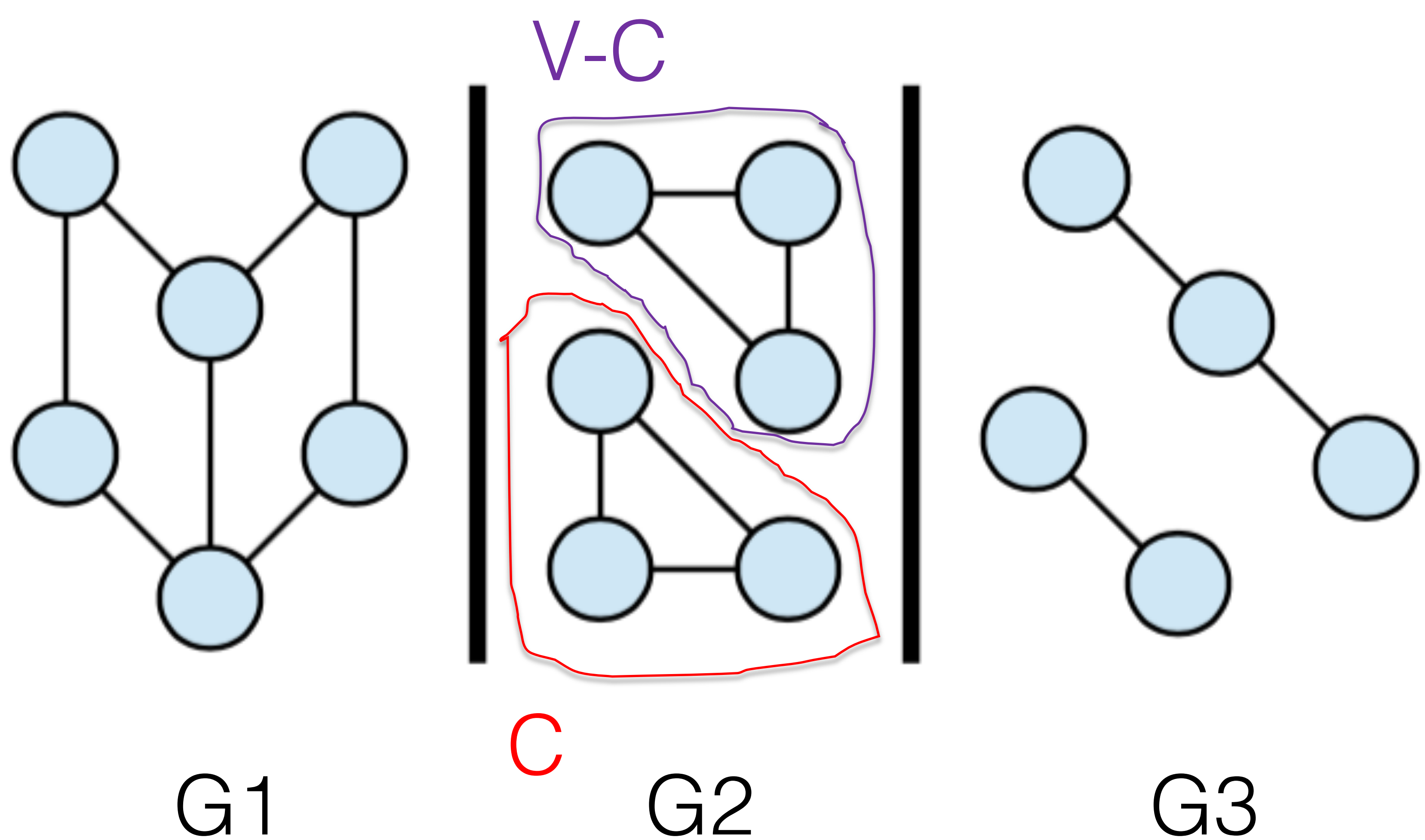
Knowing that G = (V, E), and what it means for two nodes to be connected, Let's formulate a definition for connected component of G



G1 G2 G3

Let G = (V, E) be an undirected graph.
A connected component of G is a nonempty set of nodes C (that is, $C \subseteq V$), such that

(1) For any $u, v \in C$, we have $u \leftrightarrow v$.
(2) For any $u \in C$ and $v \in V - C$, we have $u \; ! \!\leftrightarrow v$




Let G = (V, E) be an undirected graph. A connected component of G is a nonempty set of nodes C (that is, $C \subseteq V$), such that

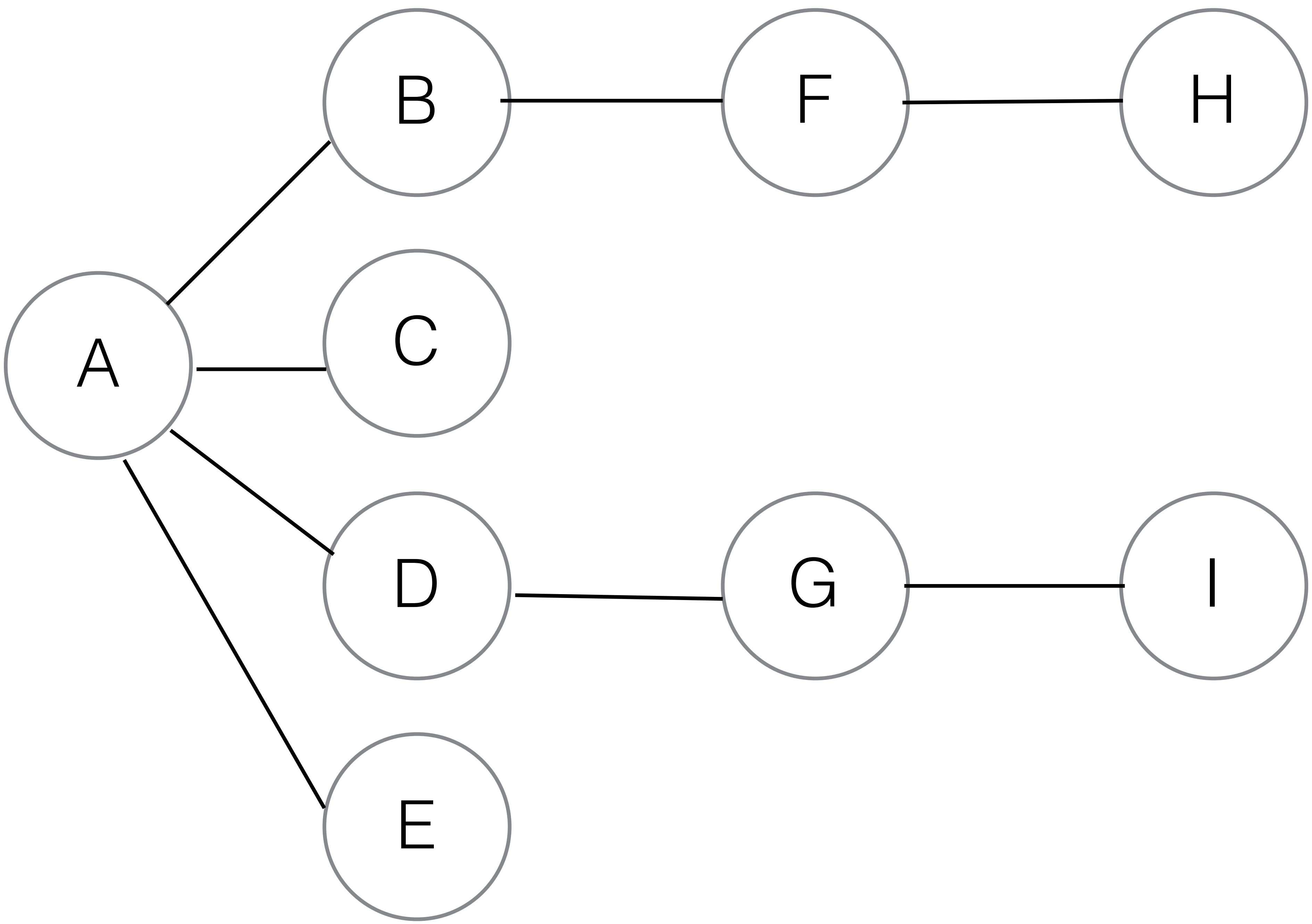
(1) For any u, v $\in$ C, we have u $\leftrightarrow$ v.

(2) For any u $\in$ C and v $\in$ V – C, we have u !$\leftrightarrow$ v

# Traversing a Graph

- There are two ways to traverse a graph:
  - **Depth-First Search (DFS)**
  - **Breadth-First Search (BFS)**
- Both will eventually reach all connected nodes
- The difference is
  - **DFS uses a stack**
  - **BFS uses a queue**

# DFS with a Stack



# DFS with a Stack (2)

- Pick a starting point - in this case vertex A, and do three things

  1. visit this vertex

  2. push it on a stack

  3. mark it visited (so you won't visit it again)

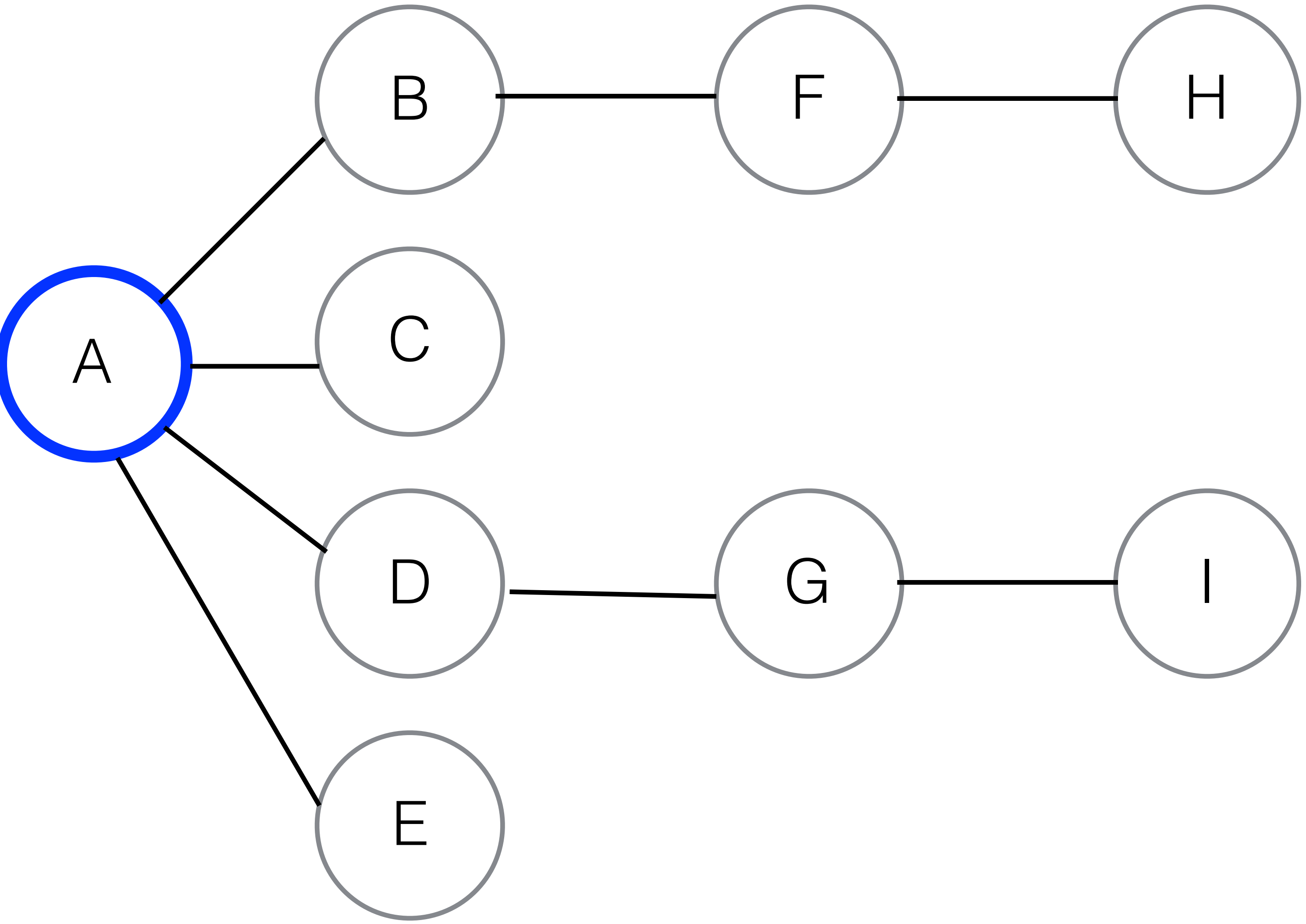
# DFS with a Stack (3)

- Pick a starting point - in this case vertex A, and do three things

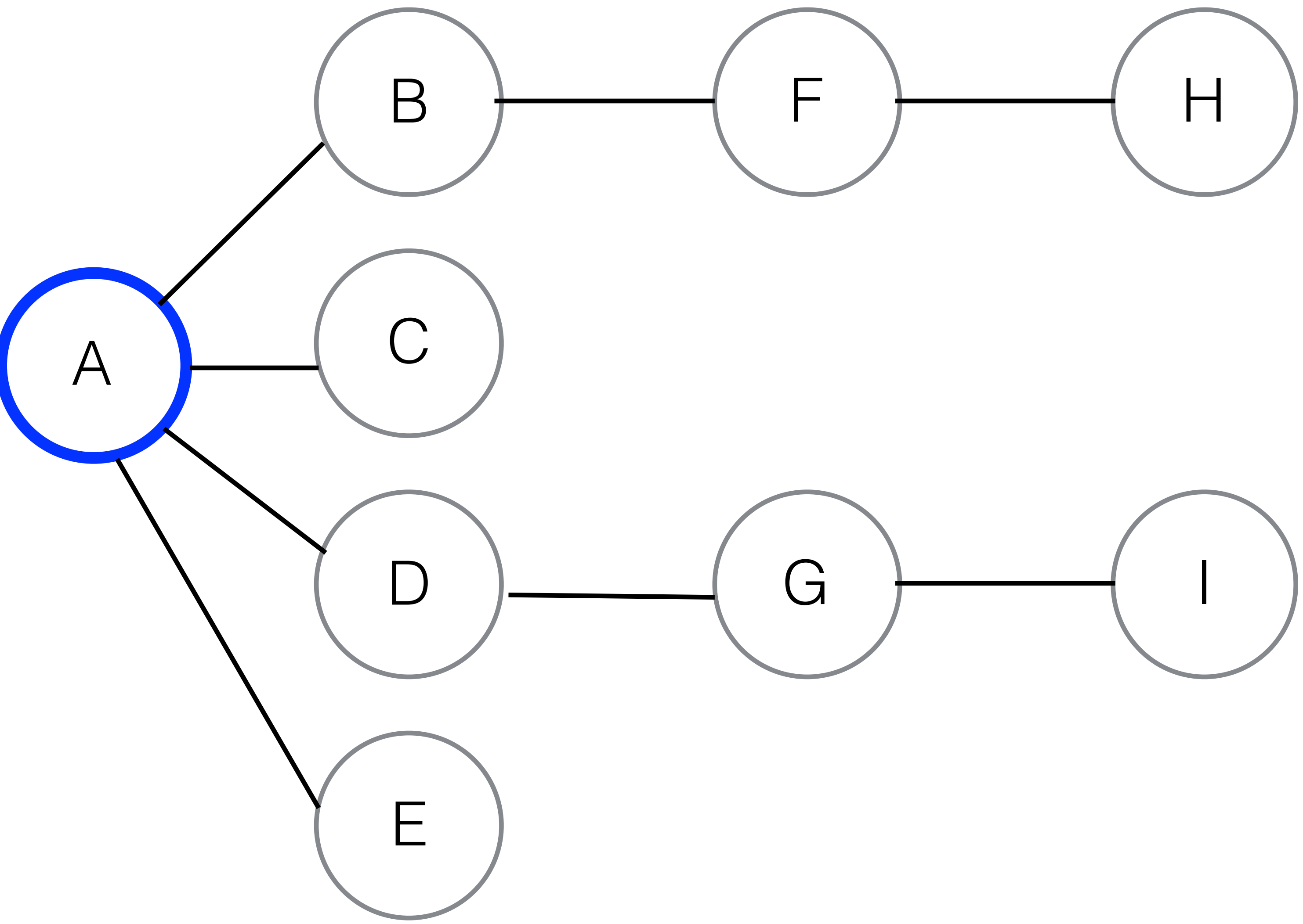
    1. visit this vertex
    2. push it on a stack
    3. mark it visited (so you won't visit it again)

Visit is abstract, just like BST

How can you mark a vertex as visited?




| **Event** | **Stack** |
|---|---|
| Visit A | A |

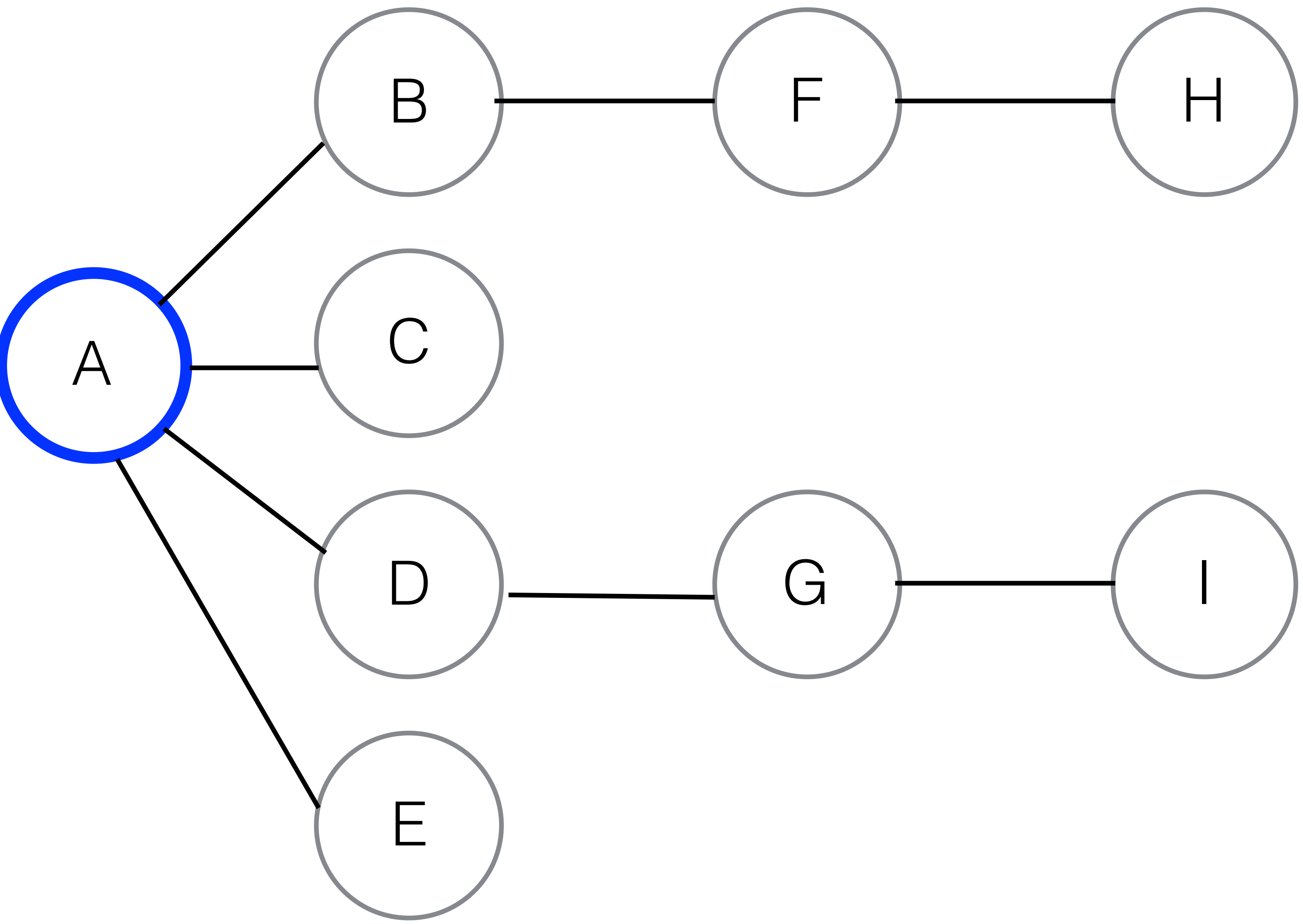



| Event | Stack |
| --- | --- |
| Visit A | A |

Next, go to a vertex adjacent to A, which hasn’t been yet visited

For this example, let’s go to B

Visit B, mark it, and push it on the stack




| Event | Stack |
|---|---|
| Visit A | A |

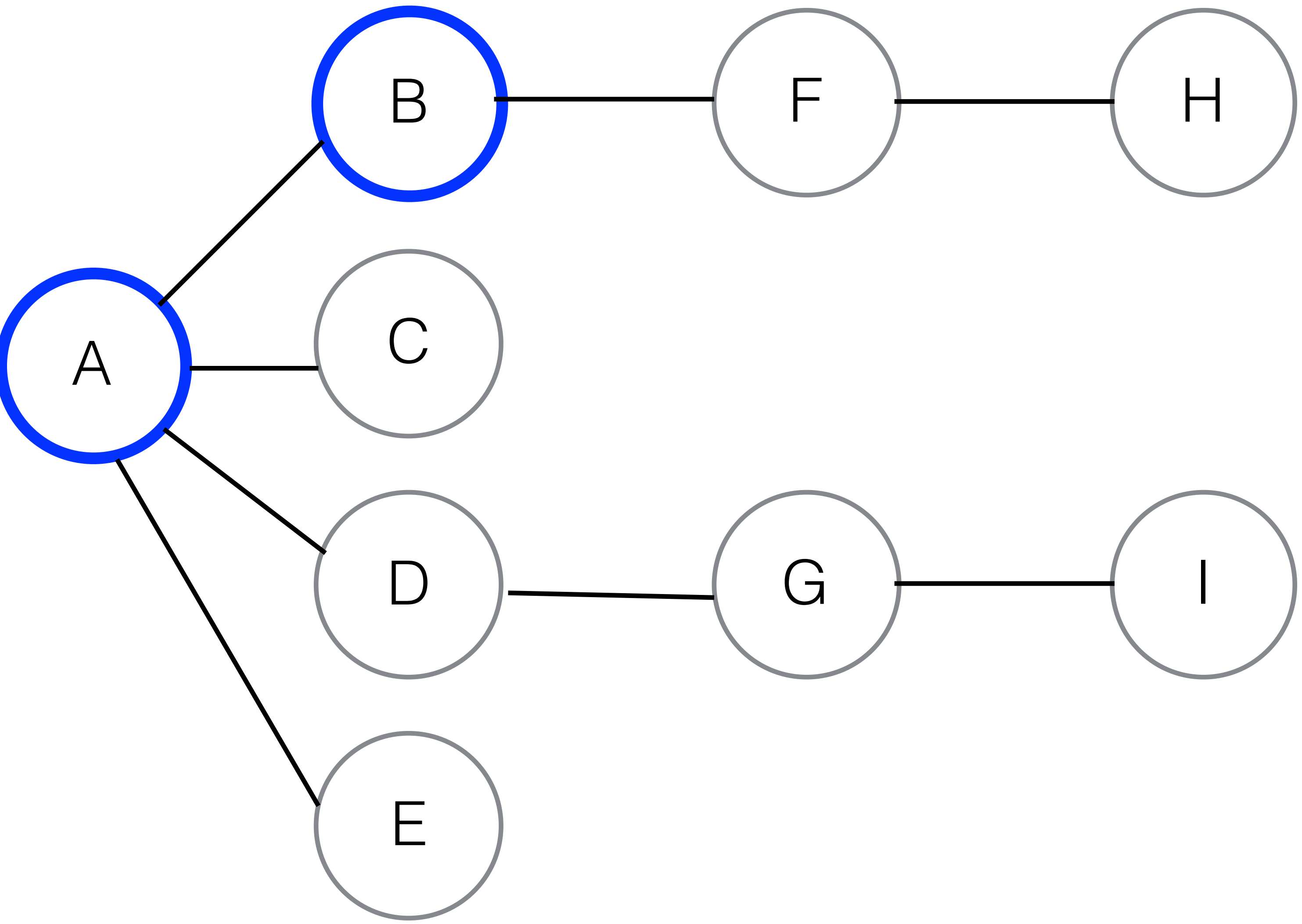
Next, go to a vertex adjacent to A, which hasn’t been yet visited

For this example, let’s go to B

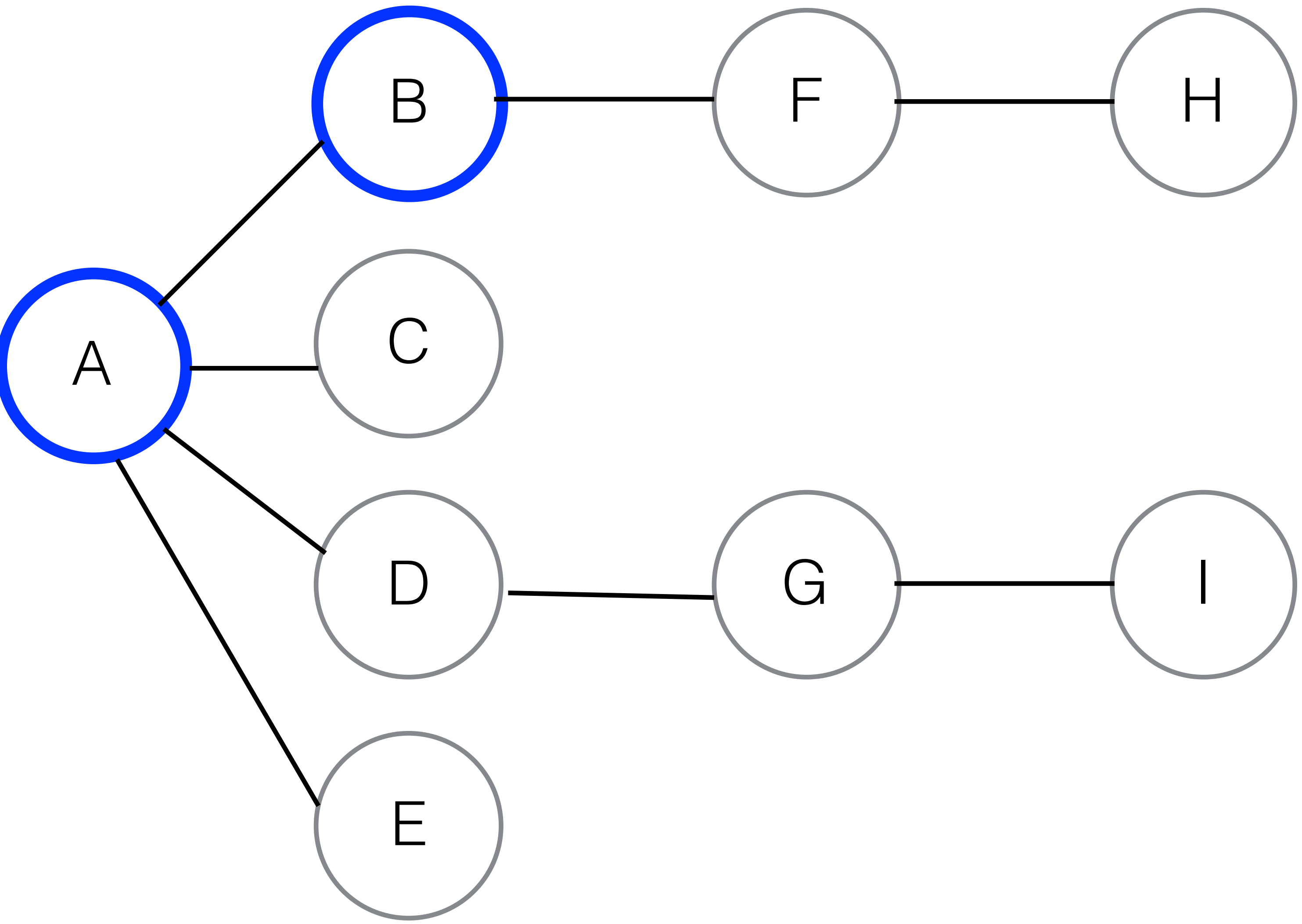
Visit B, mark it, and push it on the stack

Let’s call this **Rule 1:**

“If possible, visit an unvisited adjacent vertex, mark it, and push it on the stack”



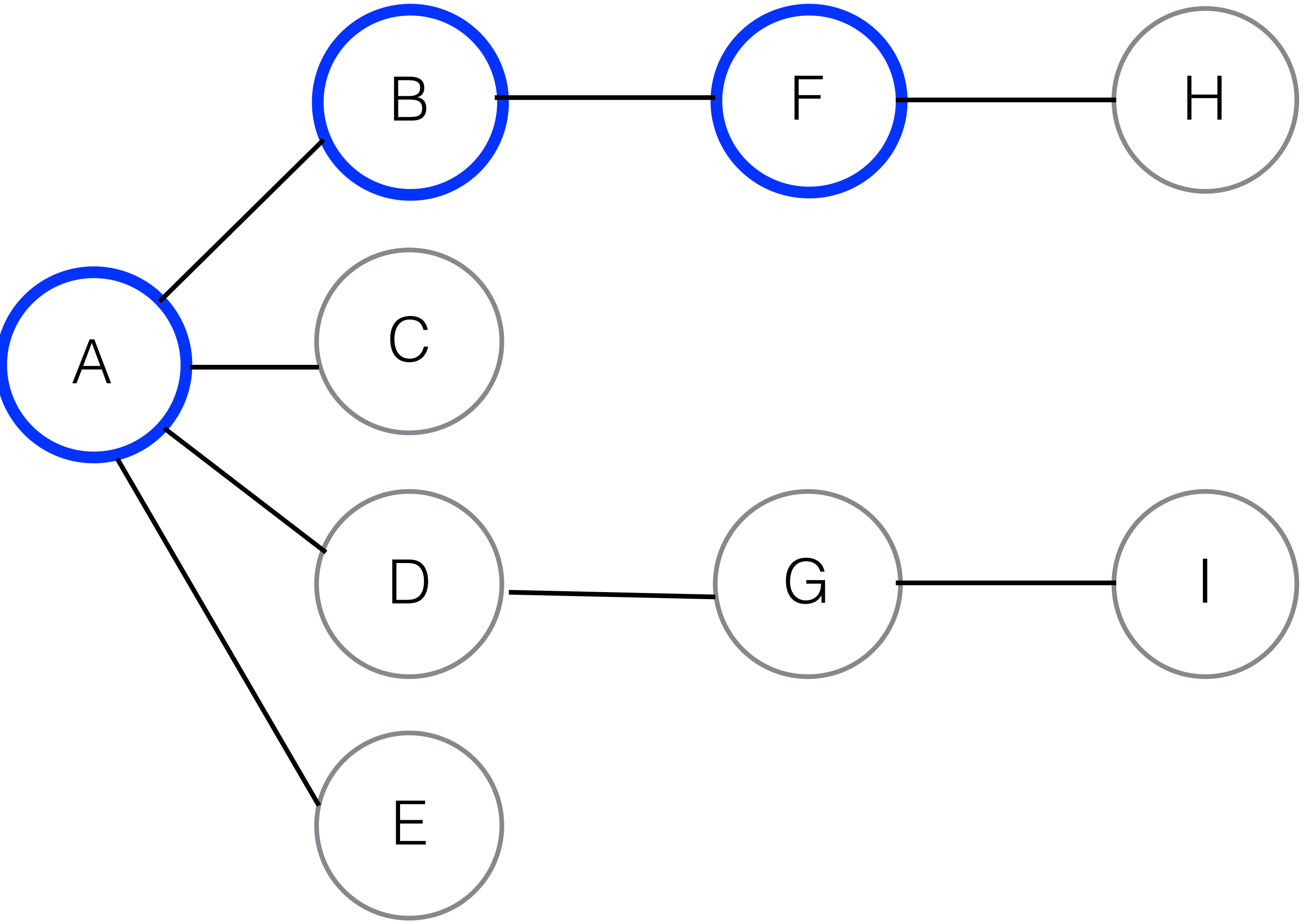

| Event | Stack |
| --- | --- |
| Visit A | A |
| Visit B | AB |



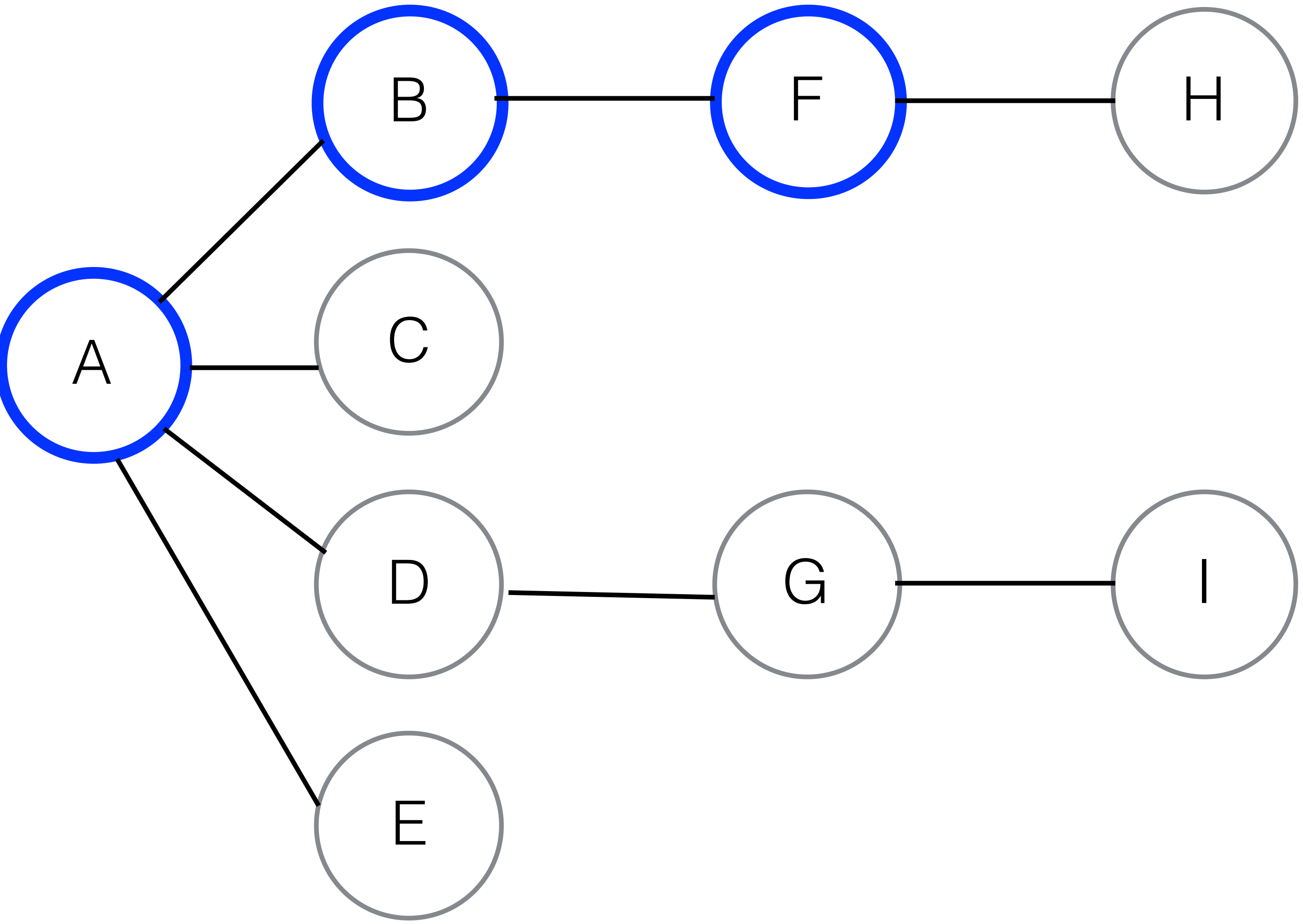

| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |

While at B, apply Rule 1 again.

"If possible, visit an unvisited adjacent vertex, mark it, and push it on the stack"

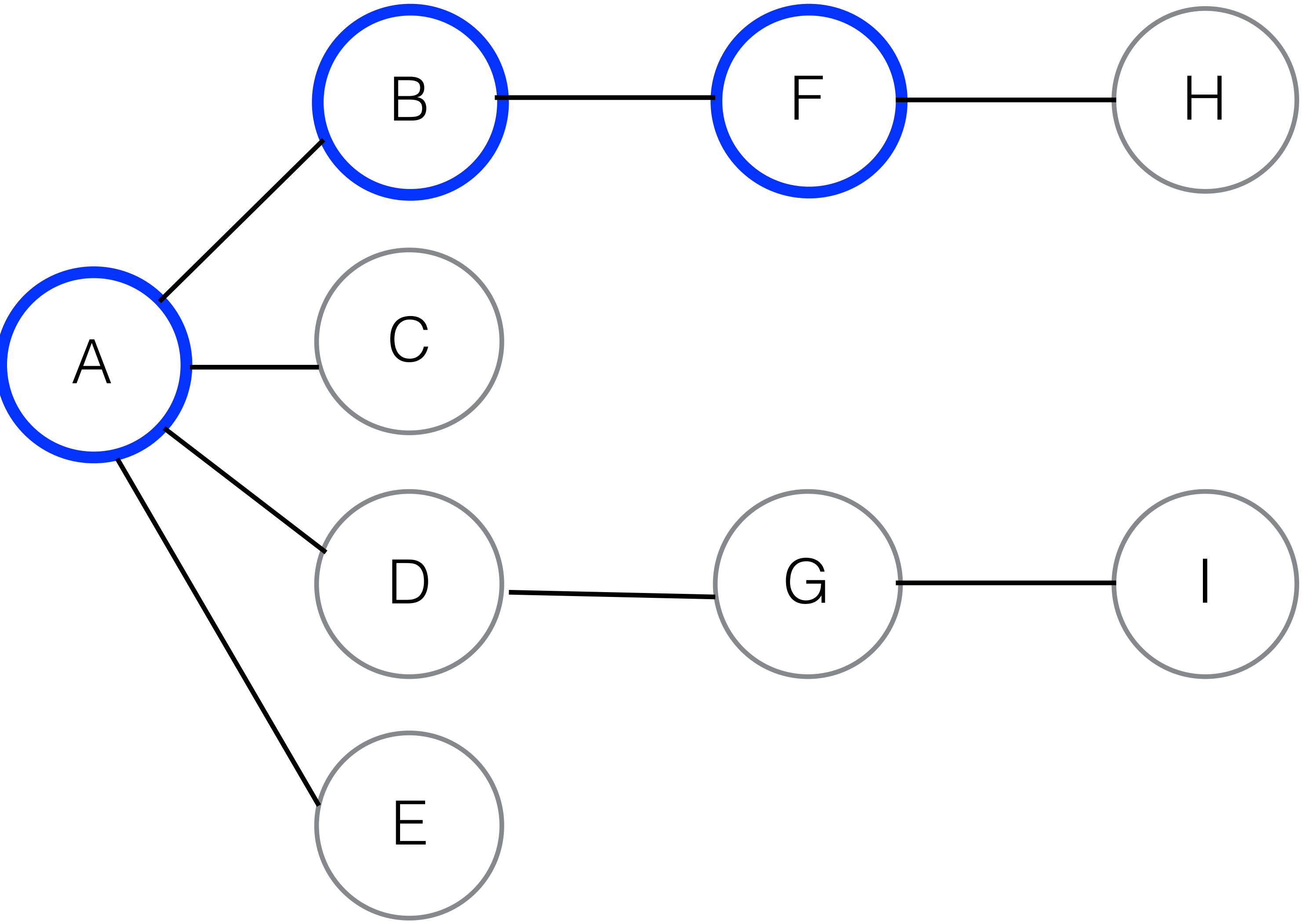



| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |



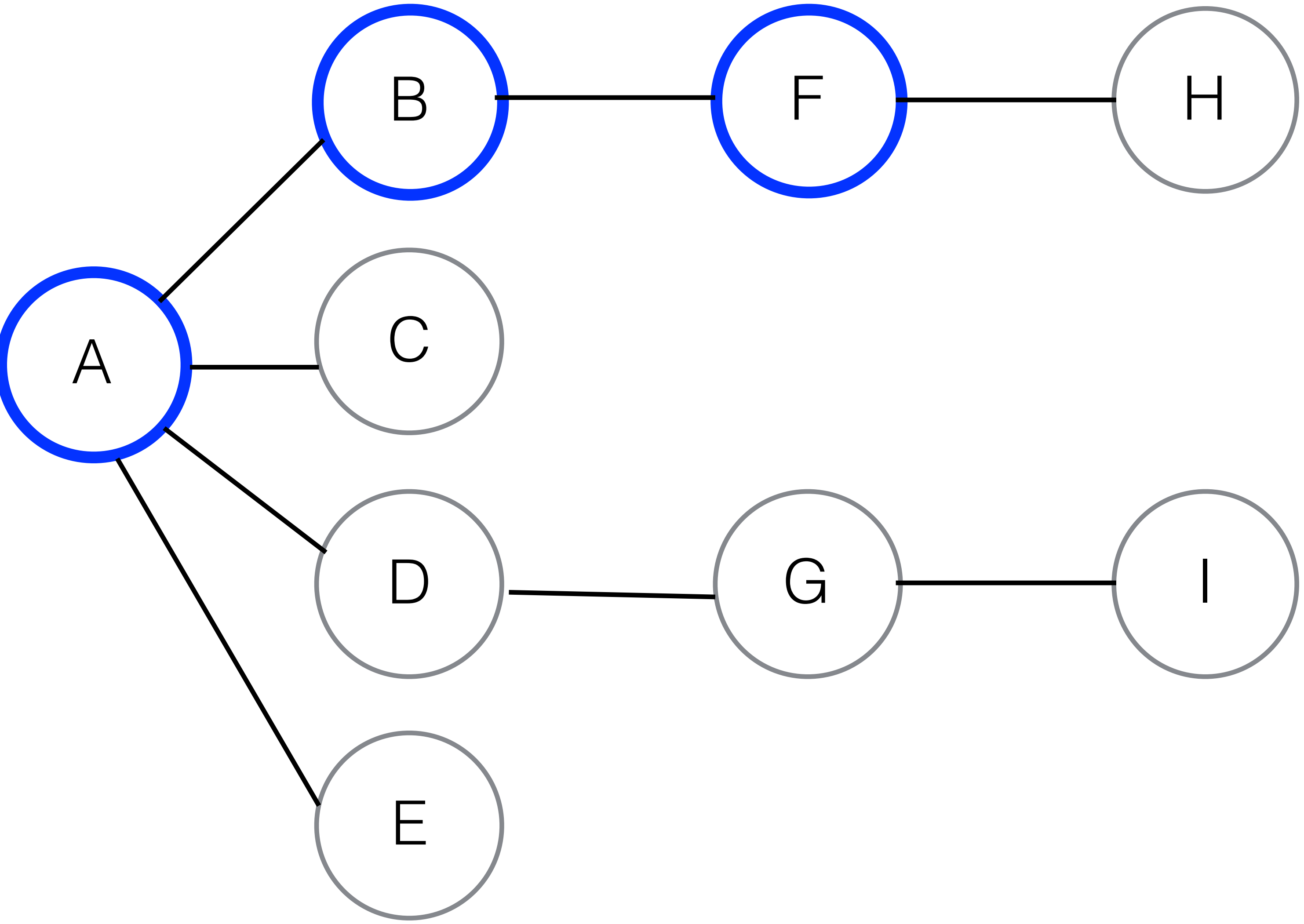

| **Event** | **Stack** |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |

What if we had picked edge “BA”?



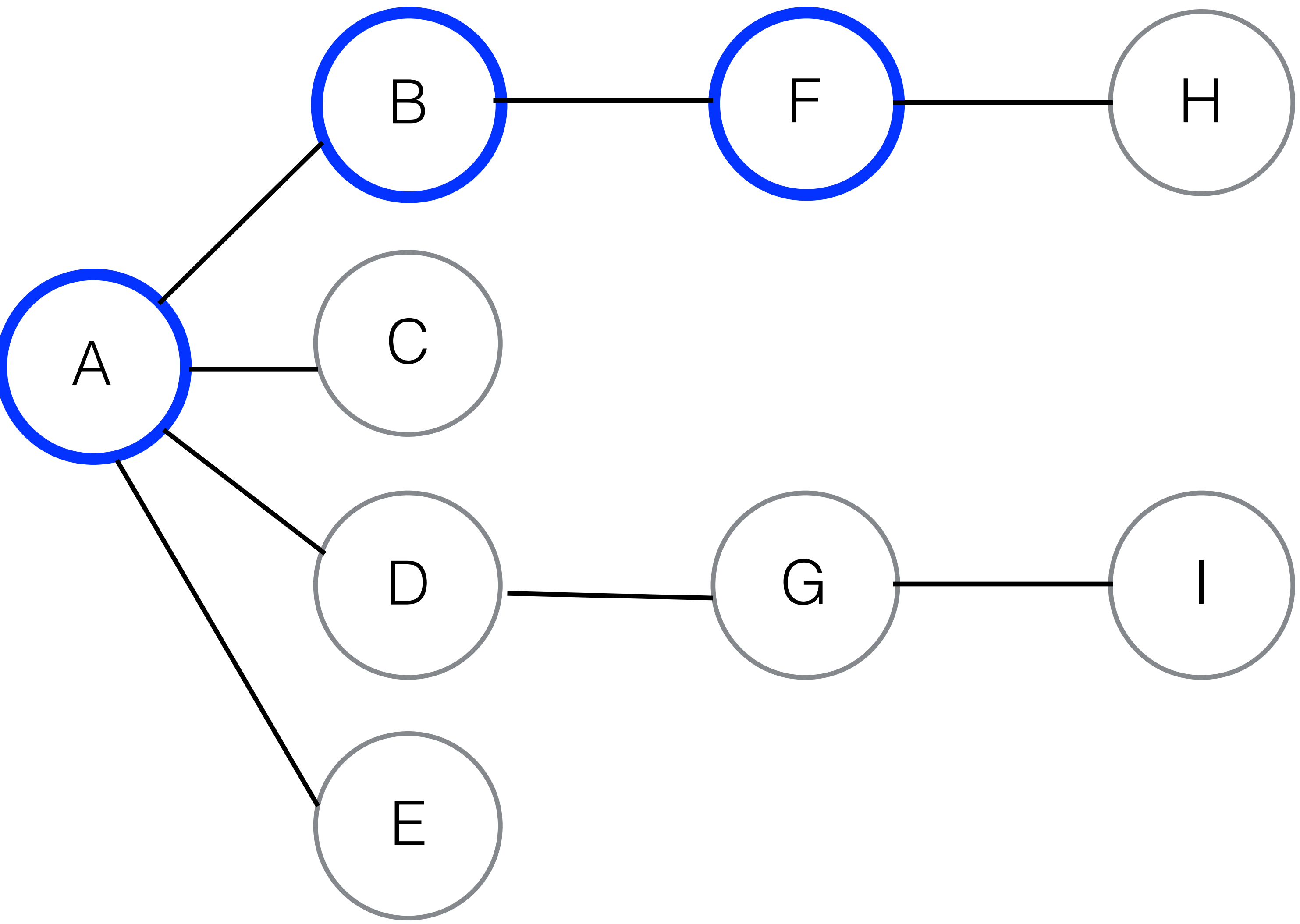

| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |

Will at some point we might pick edge “BA”?



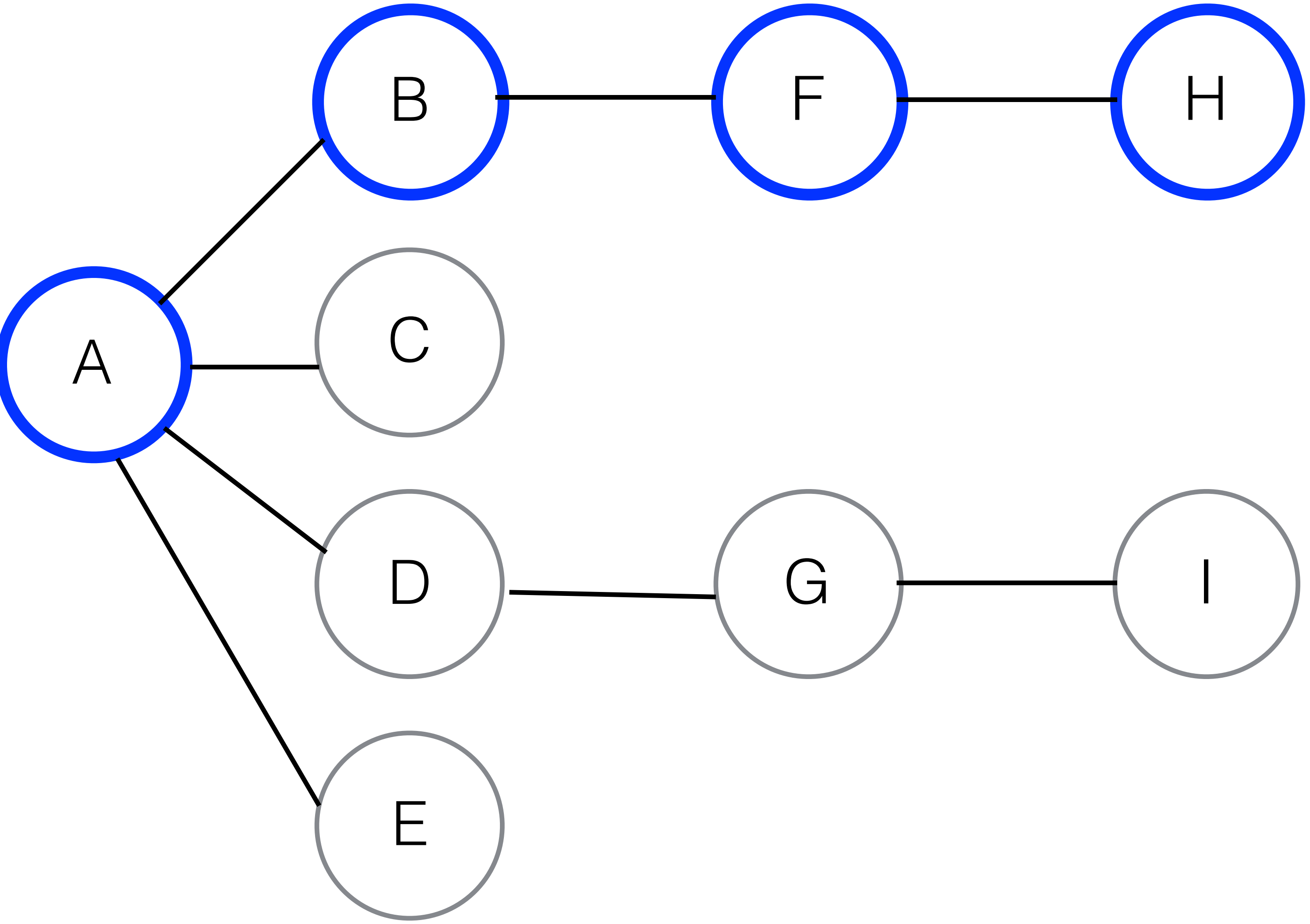

| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |

Thus you visit each vertex just once (put it in stack) but you visit each edge twice!

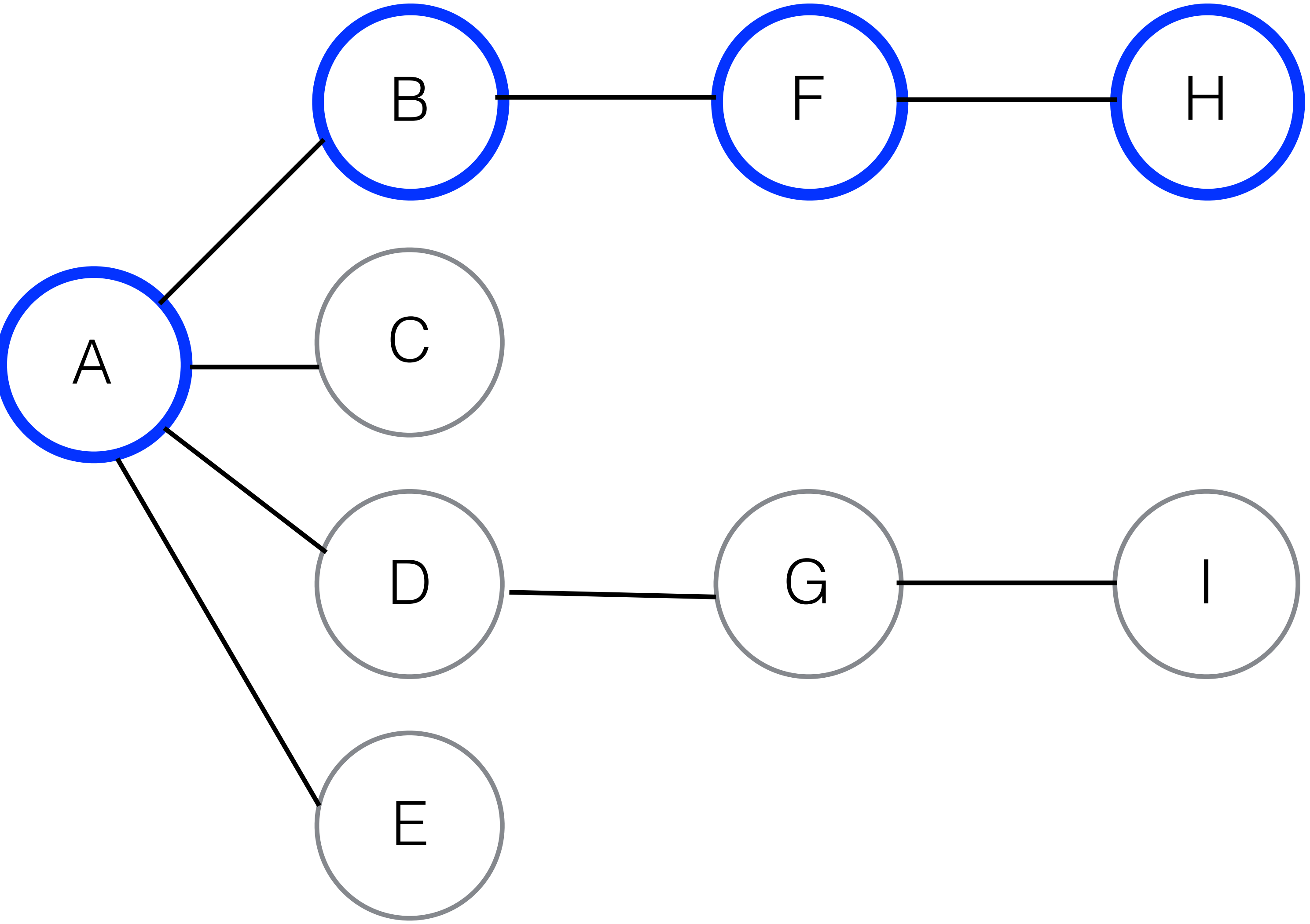



| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |

While at F, apply Rule 1 again.

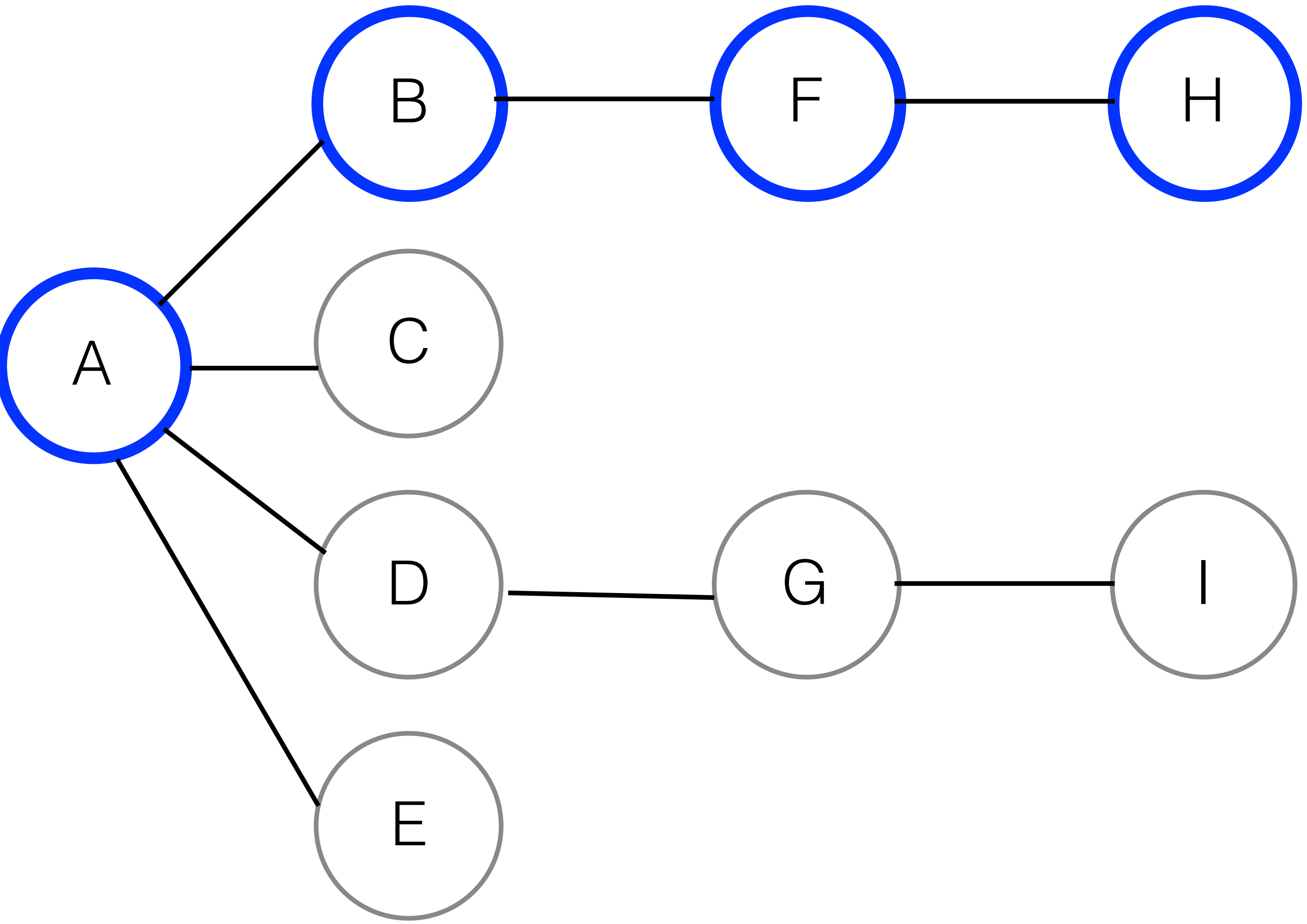



| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |




| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |

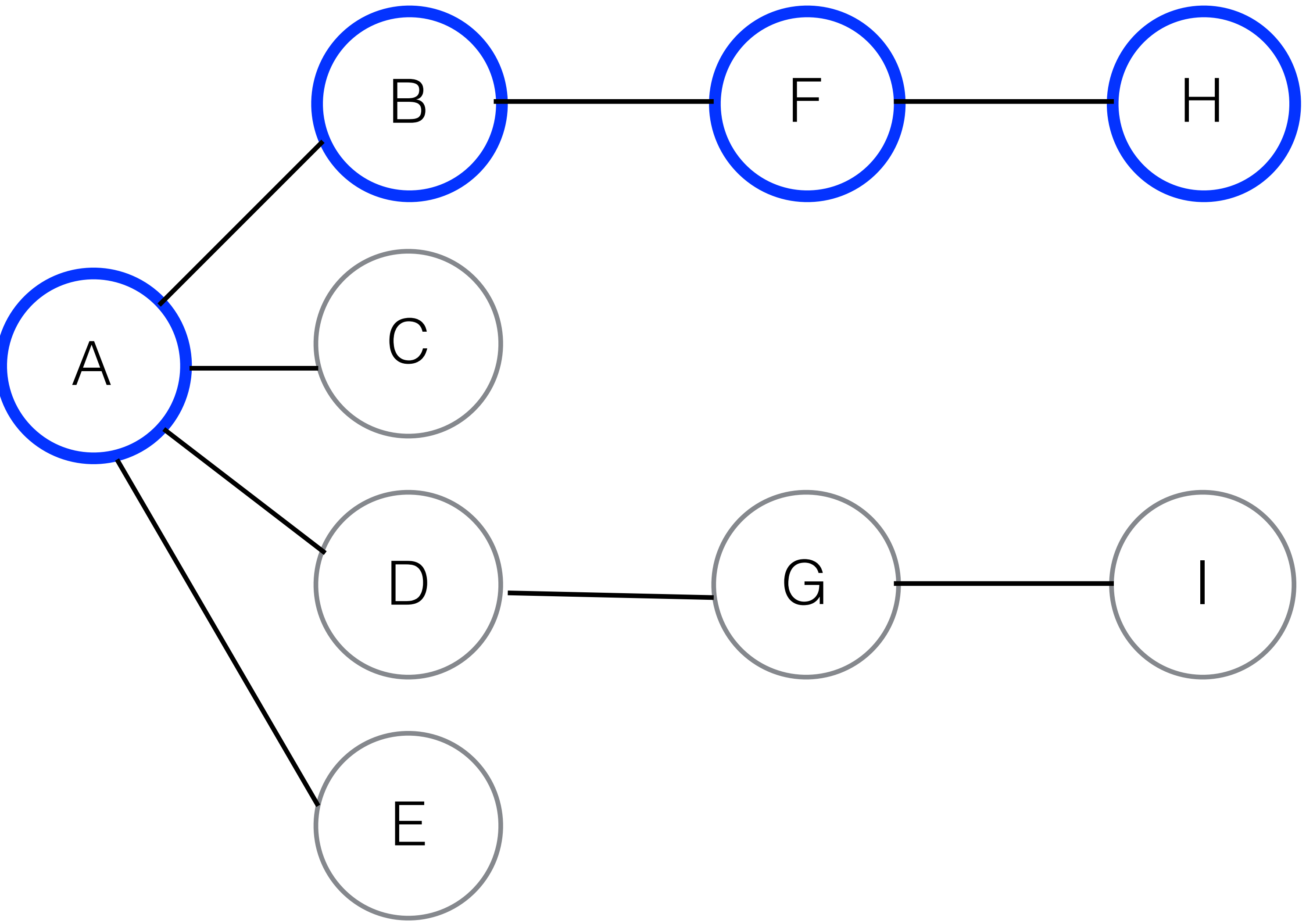
At this point (at H), there are no unvisited adjacent vertices
(HF leads back to F)
So we need to do something else



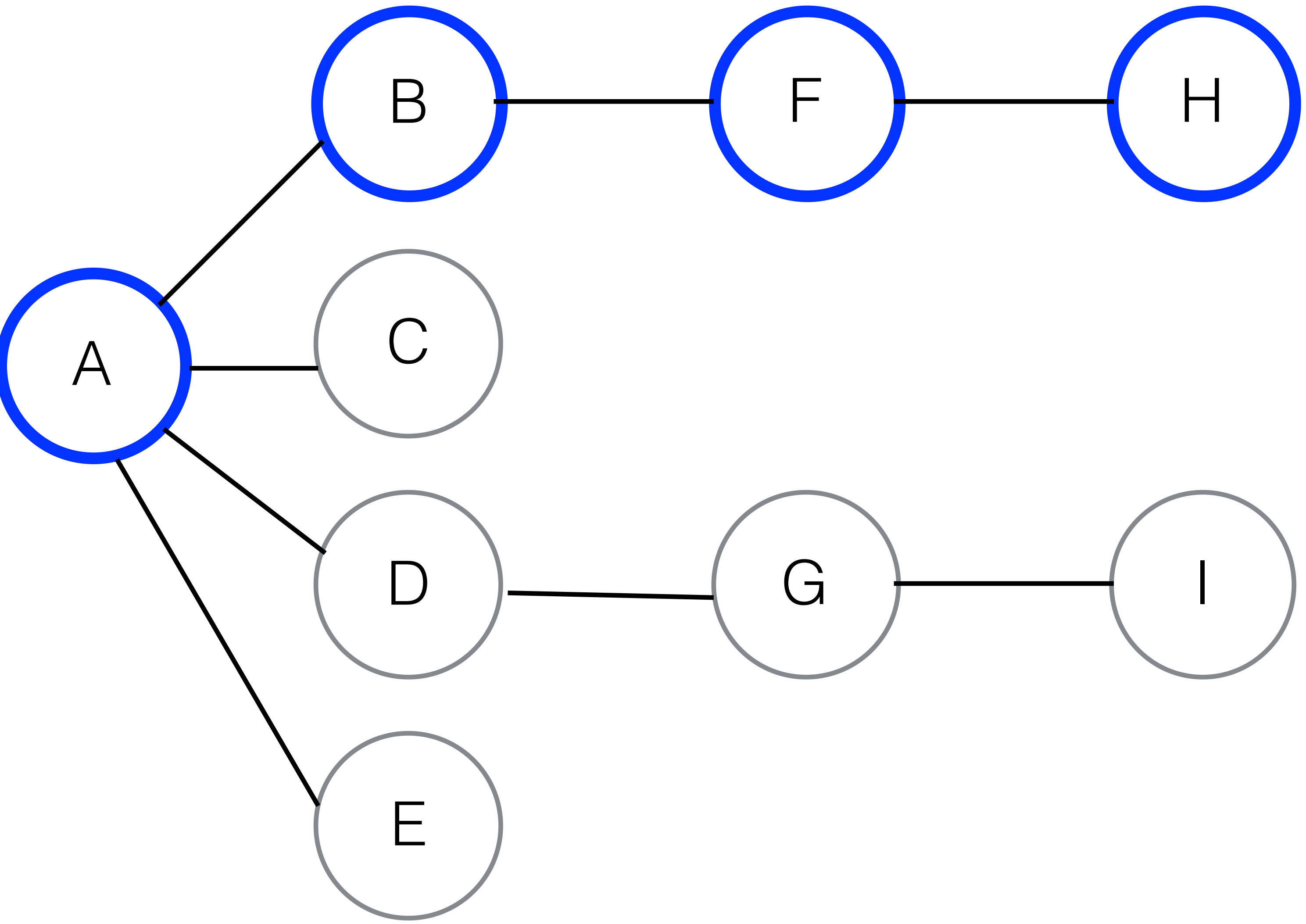

| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |

At this point (at H), there are no unvisited adjacent vertices
(HF leads back to F)
So we need to do something else

**Rule 2:**
"If you cannot follow Rule 1, then, if possible, pop a vertex off the stack"



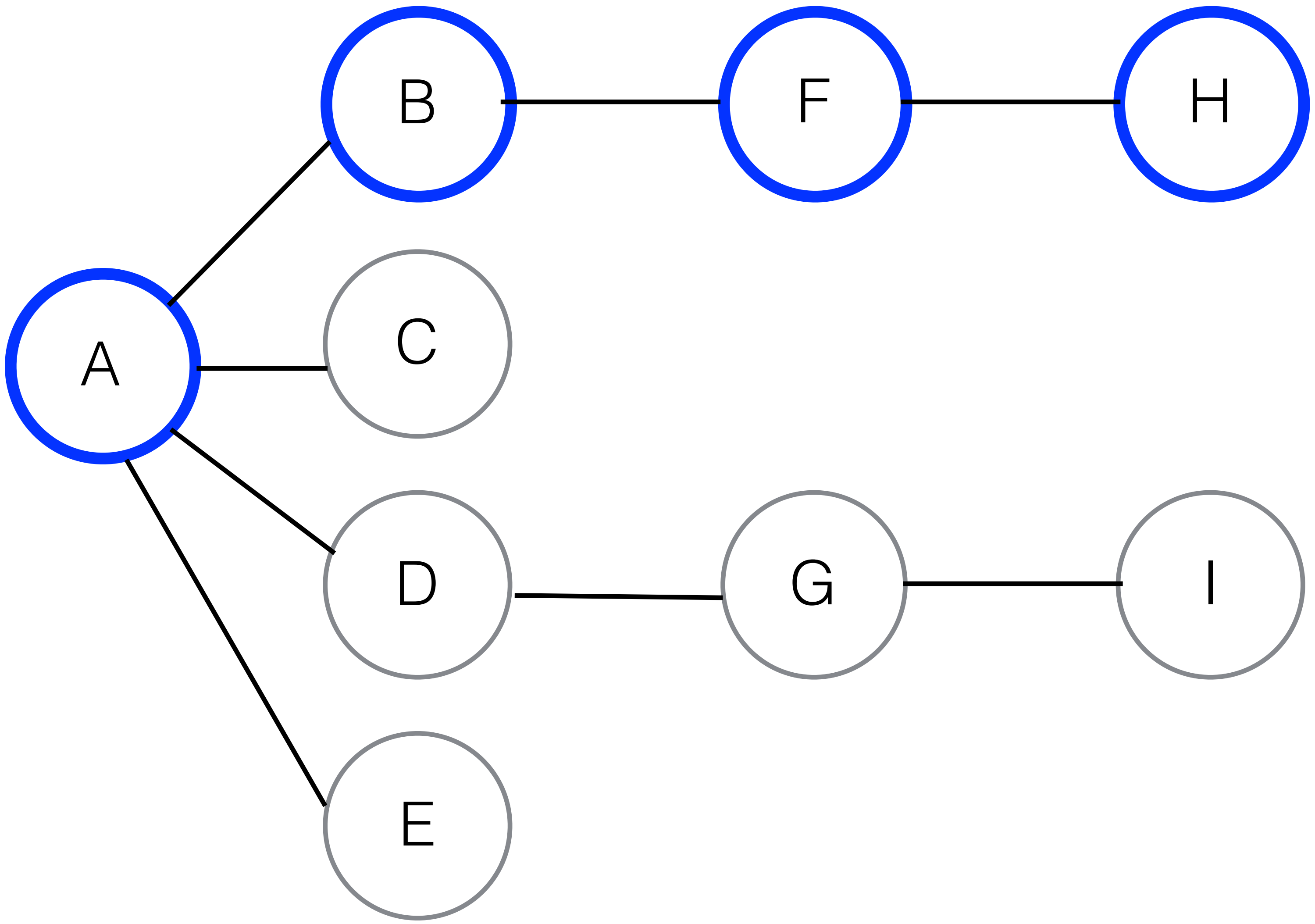

| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |



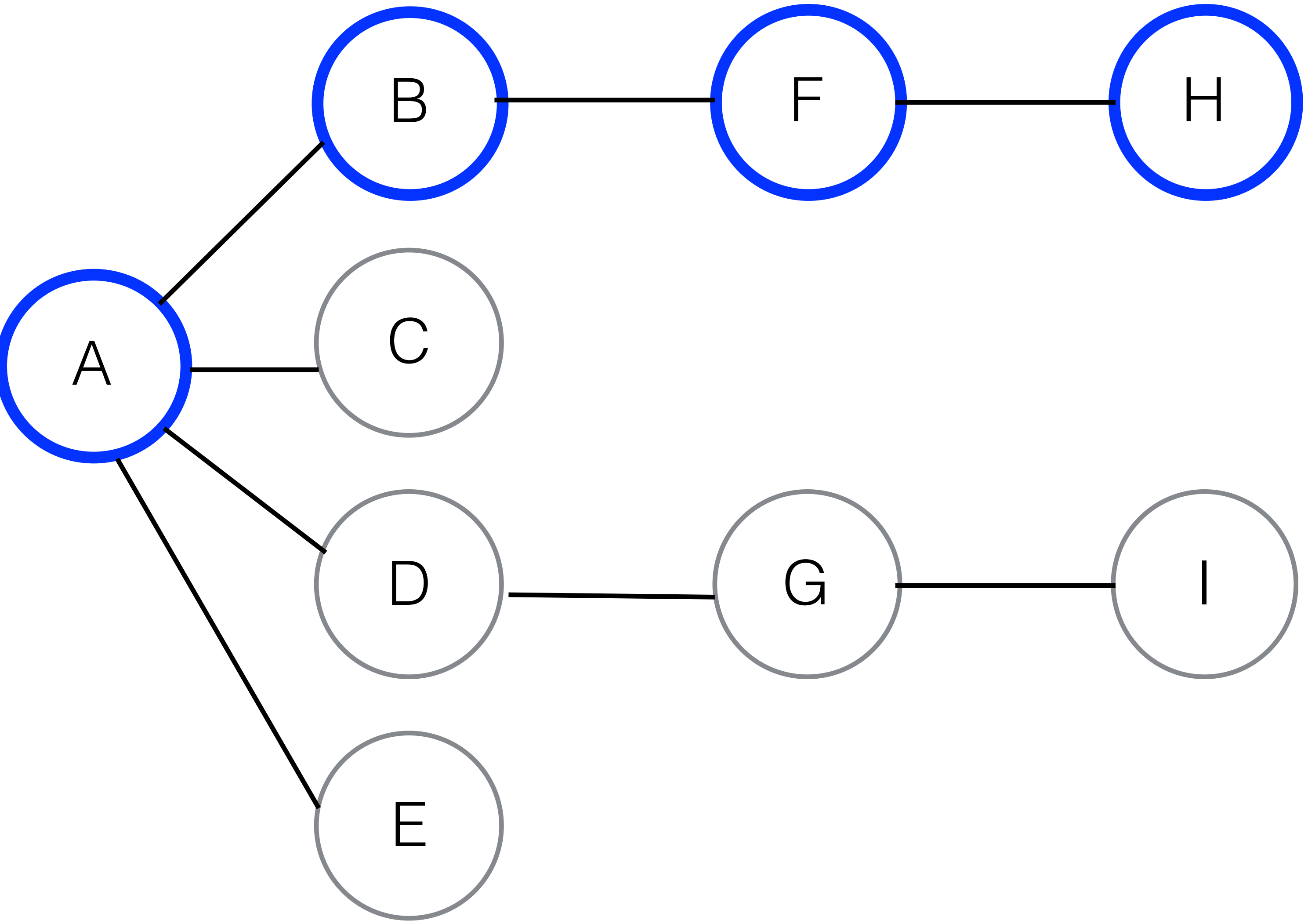

| Event | Stack |
| --- | --- |
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |

We are back at F

No more unvisited adjacent vertices, so pop it off, too

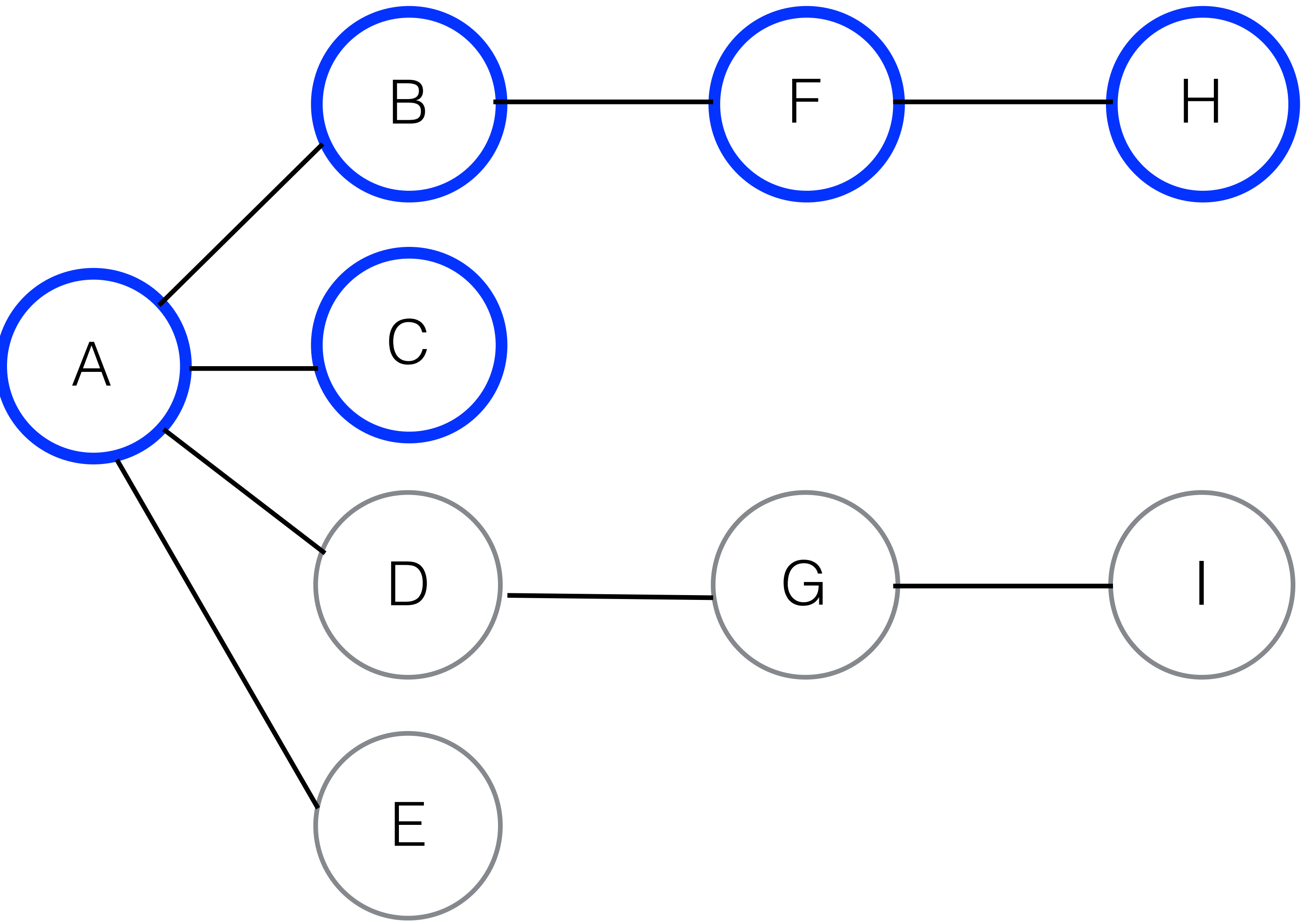



| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |

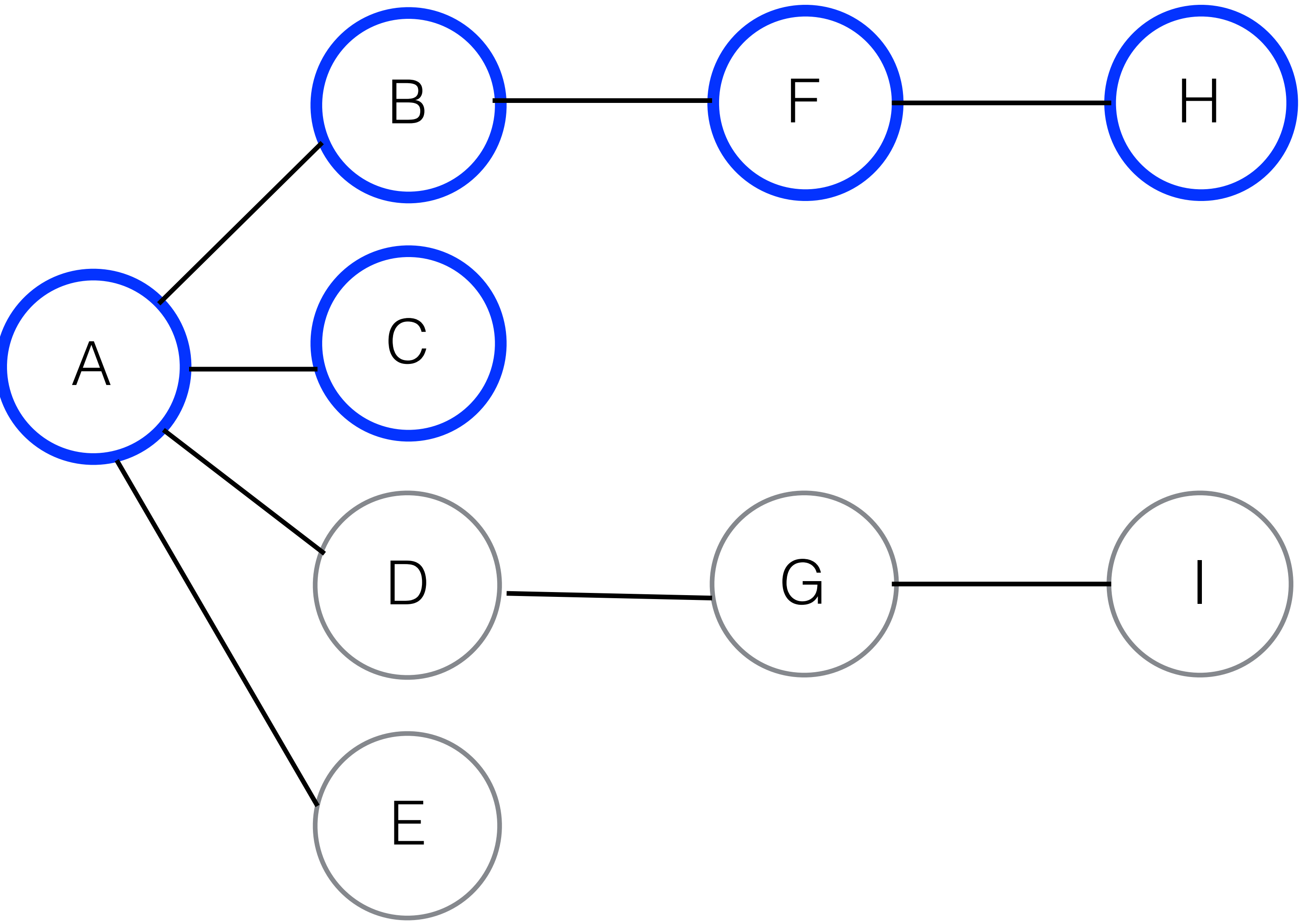



| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |

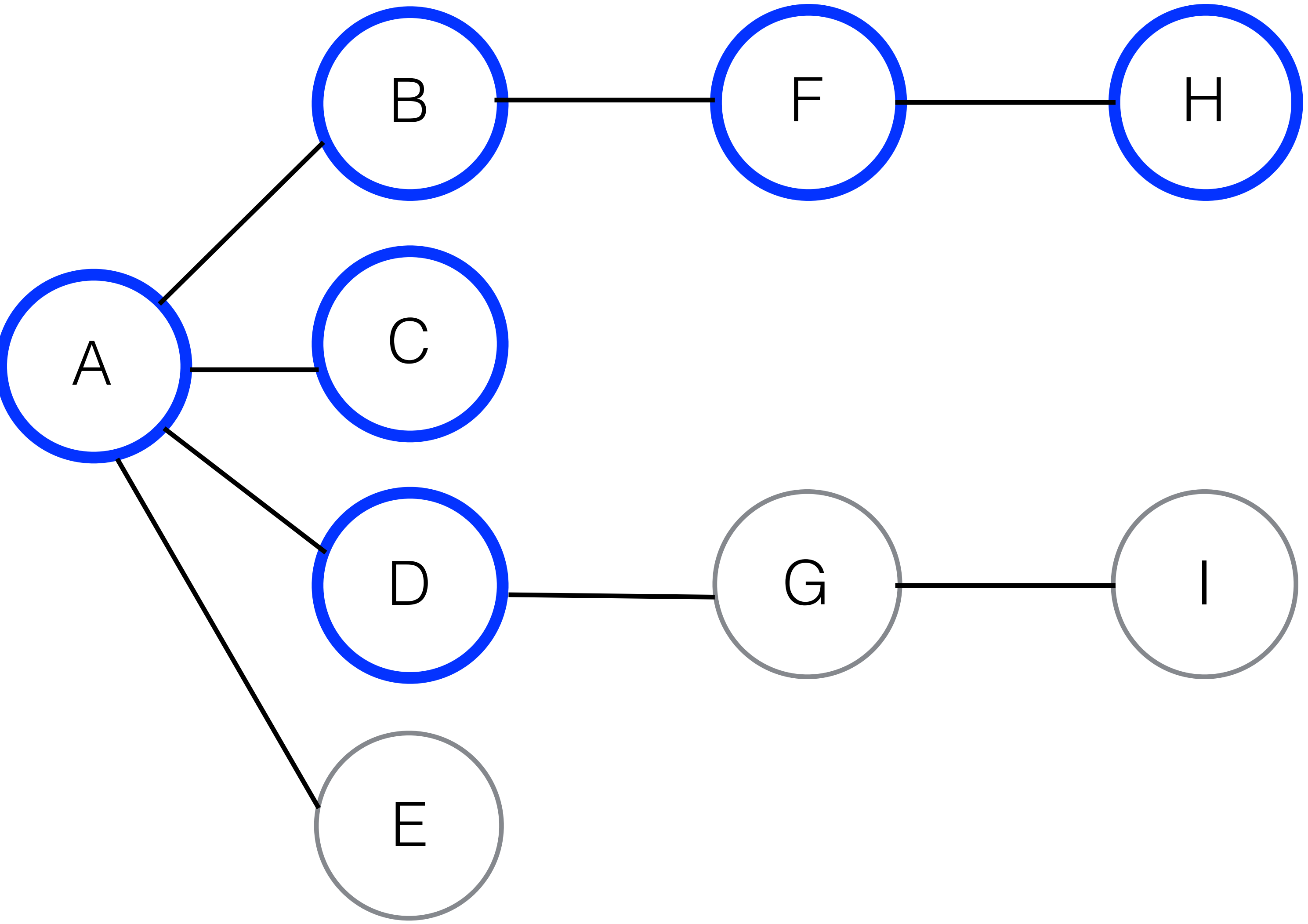
- We are back at A
- Pick the next adjacent vertex and repeat



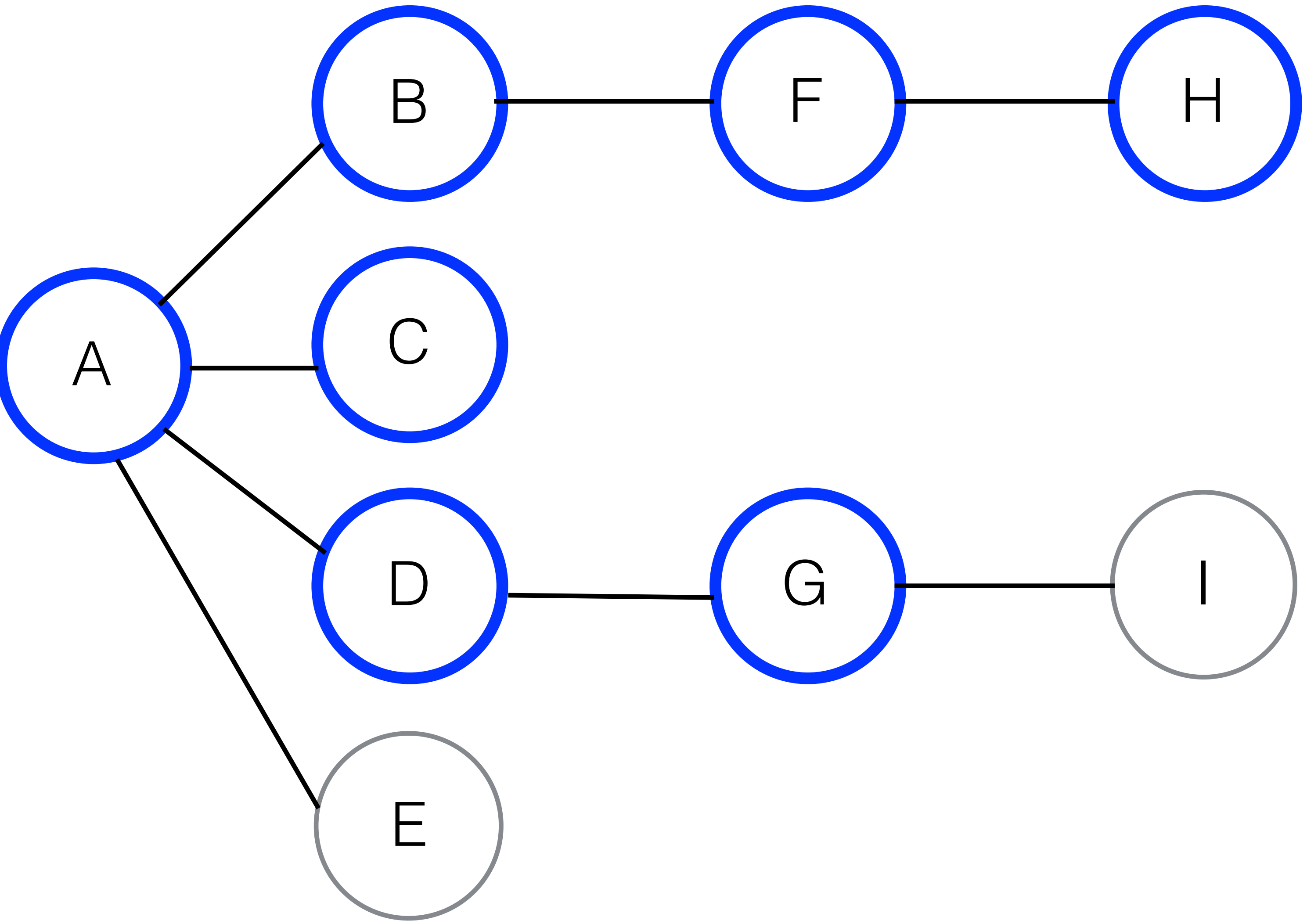

| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |
| Visit C | AC |



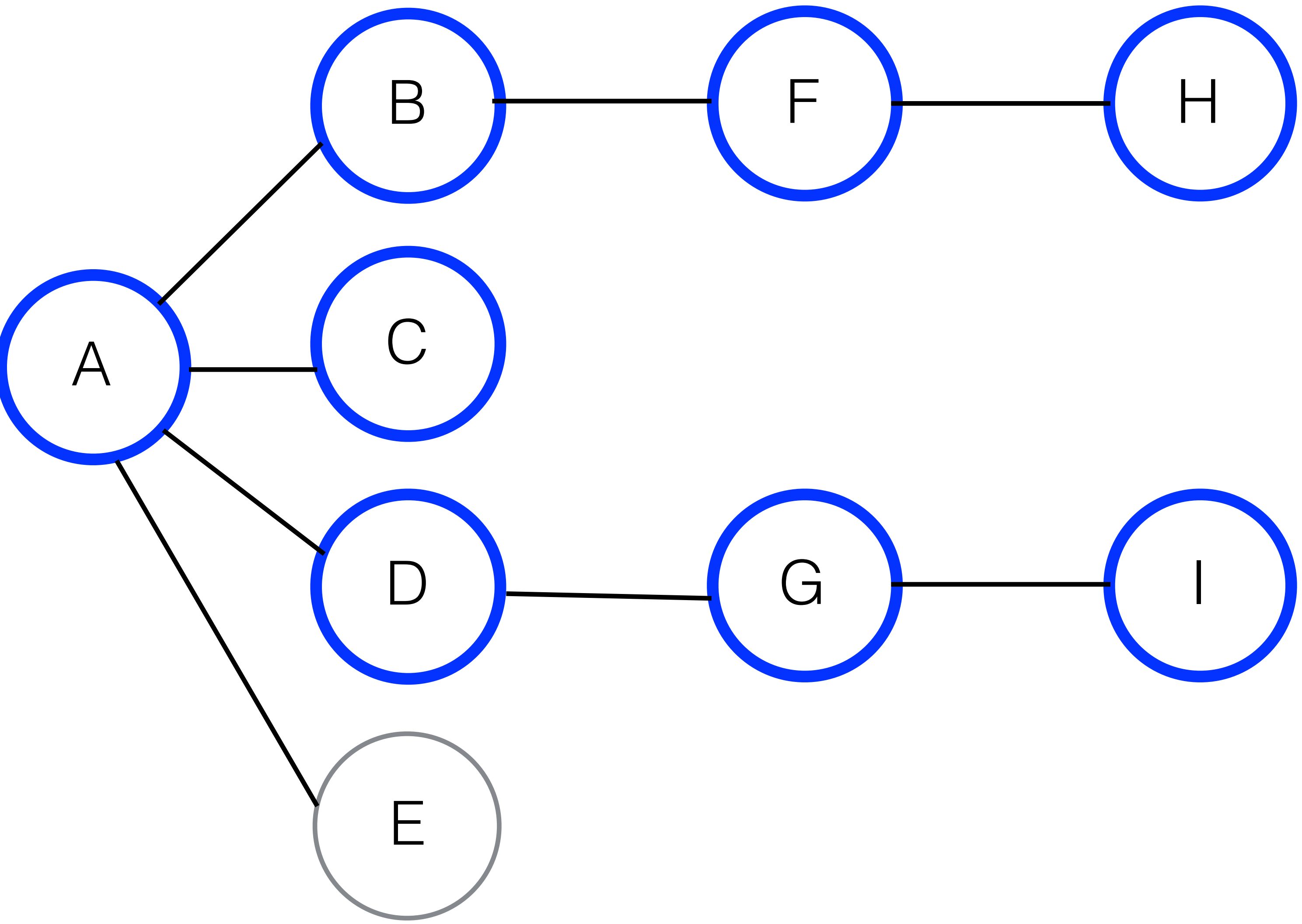

| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |
| Visit C | AC |
| Pop C | A |



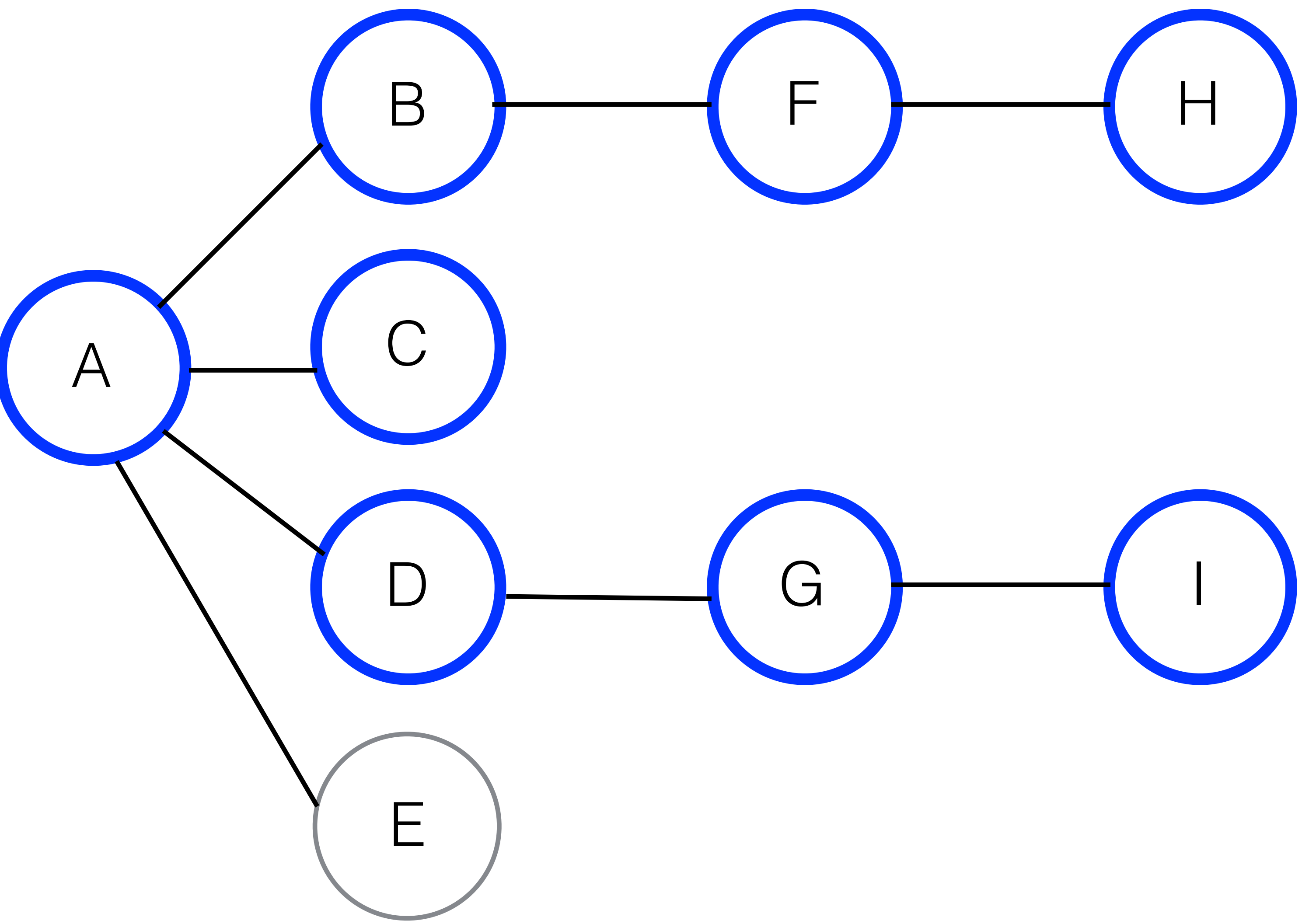

| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |
| Visit C | AC |
| Pop C | A |
| Visit D | AD |



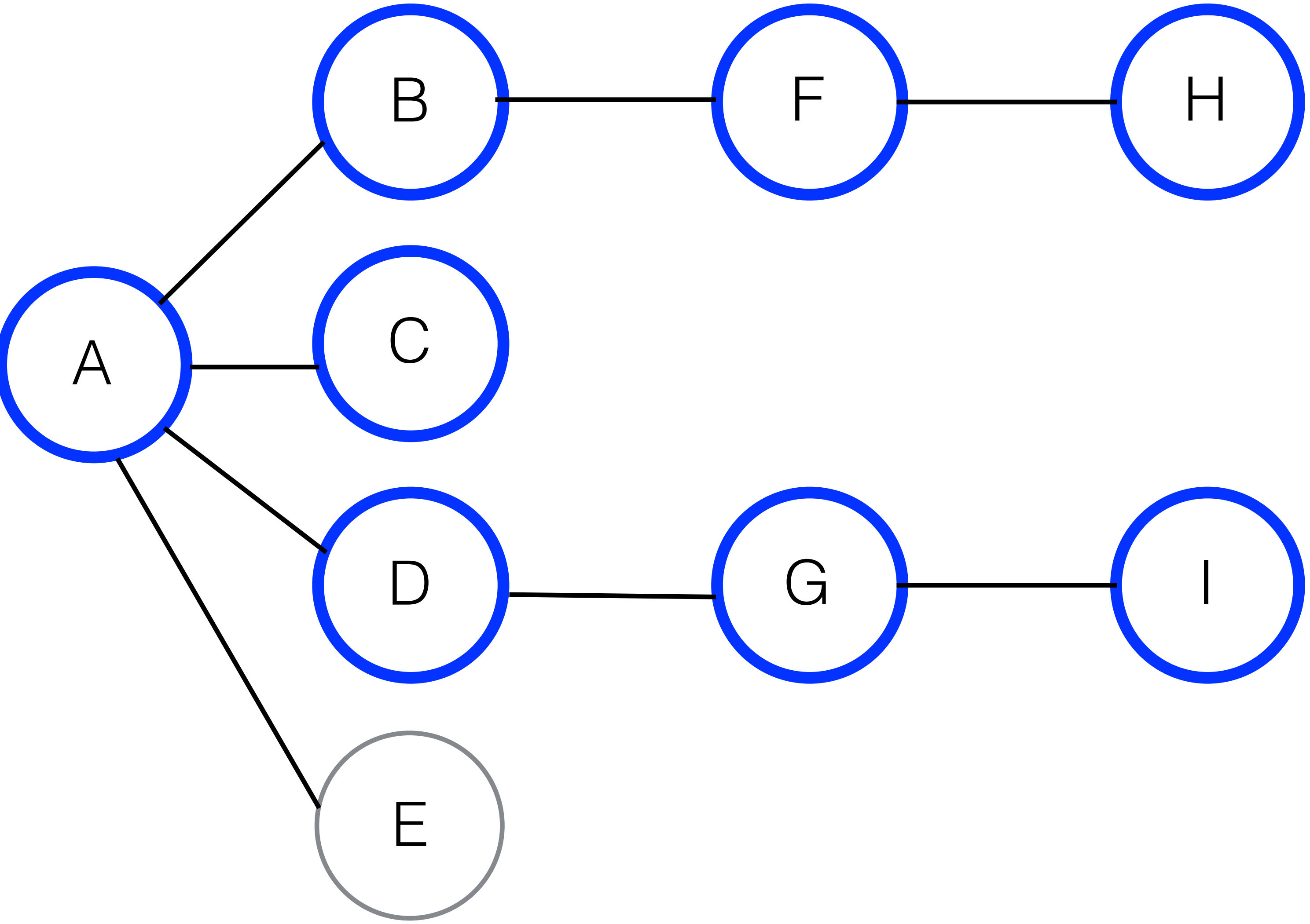

| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |
| Visit C | AC |
| Pop C | A |
| Visit D | AD |
| Visit G | ADG |



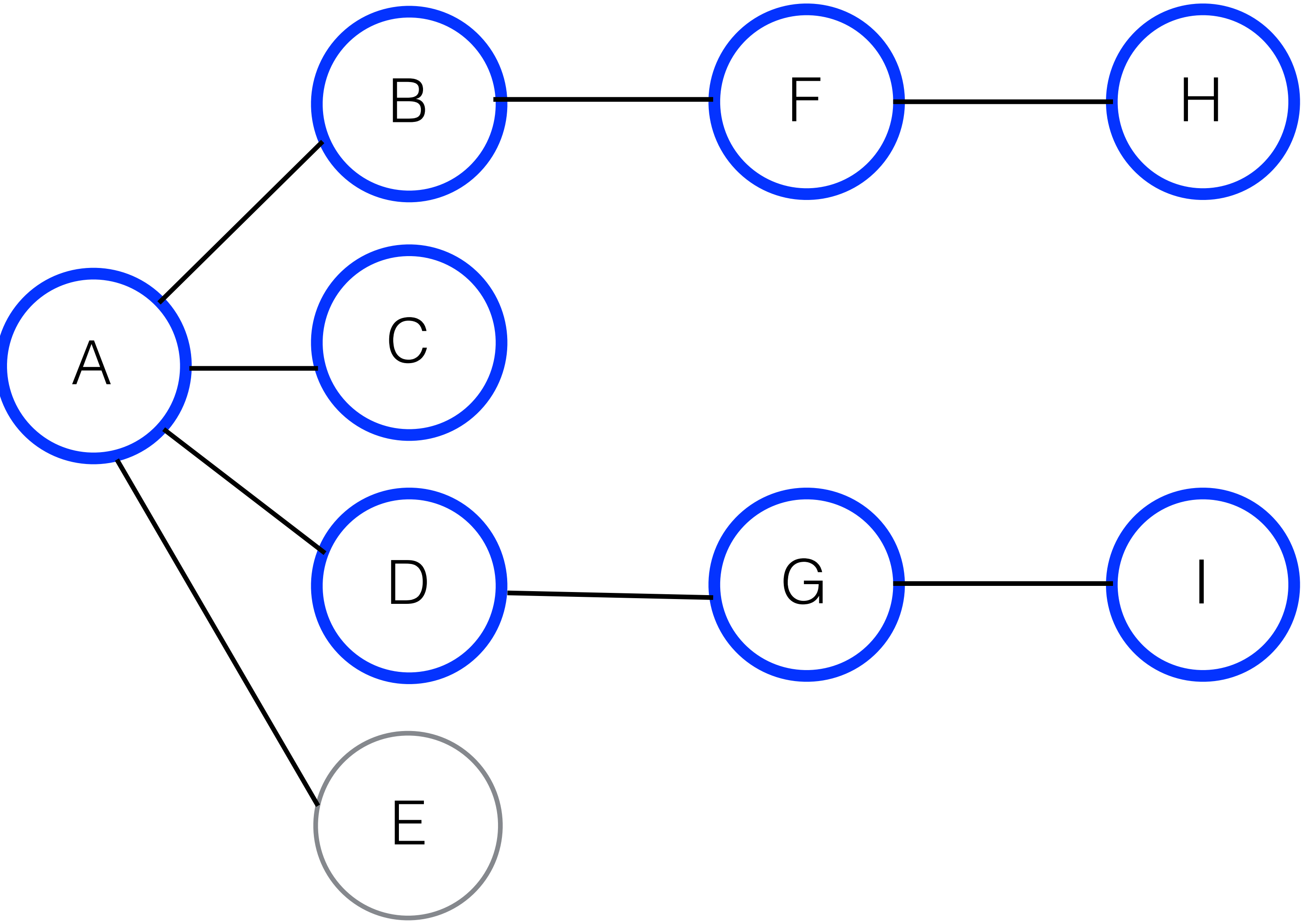

| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |
| Visit C | AC |
| Pop C | A |
| Visit D | AD |
| Visit G | ADG |
| Visit I | ADGI |



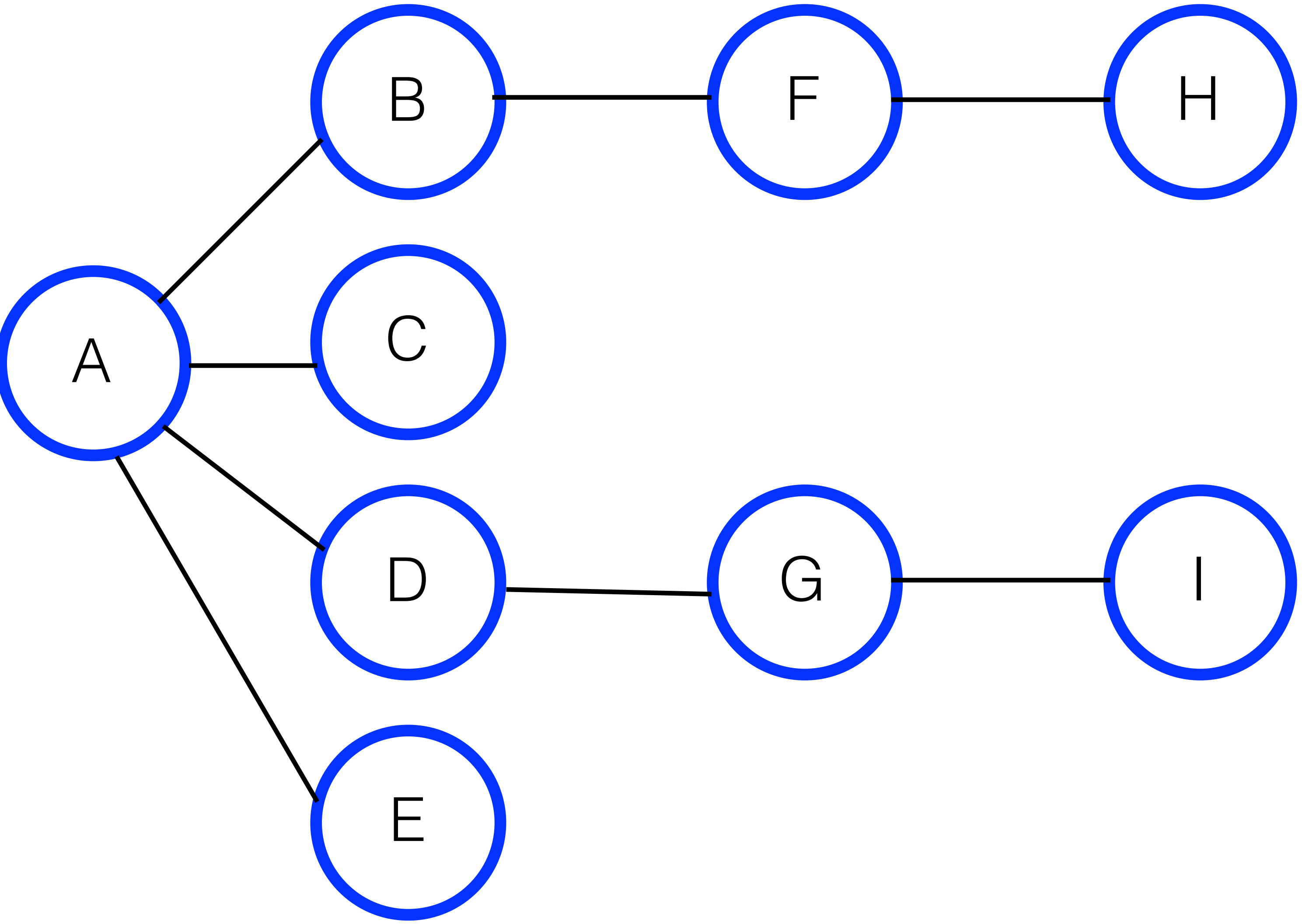

| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |
| Visit C | AC |
| Pop C | A |
| Visit D | AD |
| Visit G | ADG |
| Visit I | ADGI |
| Pop I | ADG |



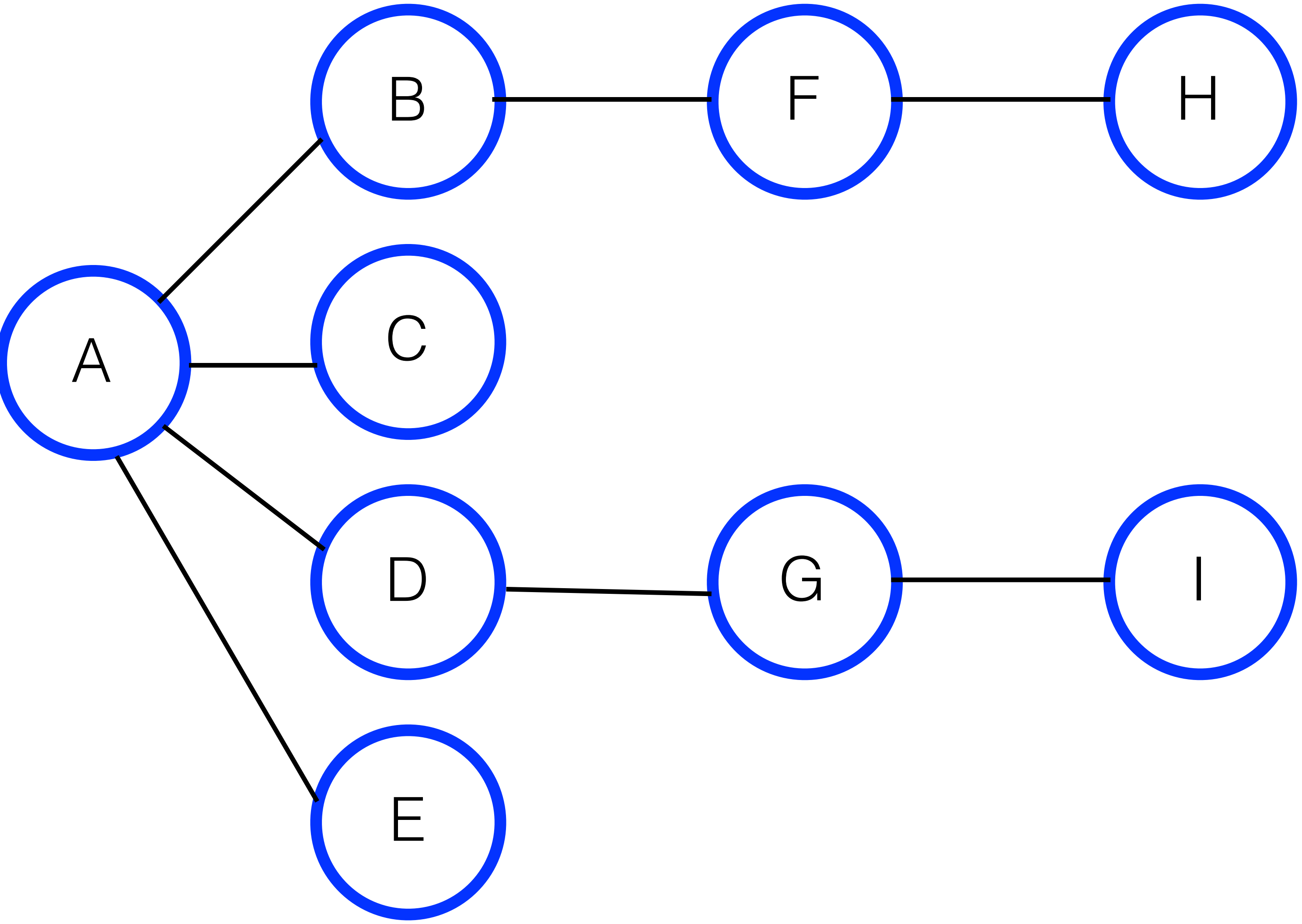

| Event | Stack |
| --- | --- |
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |
| Visit C | AC |
| Pop C | A |
| Visit D | AD |
| Visit G | ADG |
| Visit I | ADGI |
| Pop I | ADG |
| Pop G | AD |




| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |
| Visit C | AC |
| Pop C | A |
| Visit D | AD |
| Visit G | ADG |
| Visit I | ADGI |
| Pop I | ADG |
| Pop G | AD |
| Pop D | A |

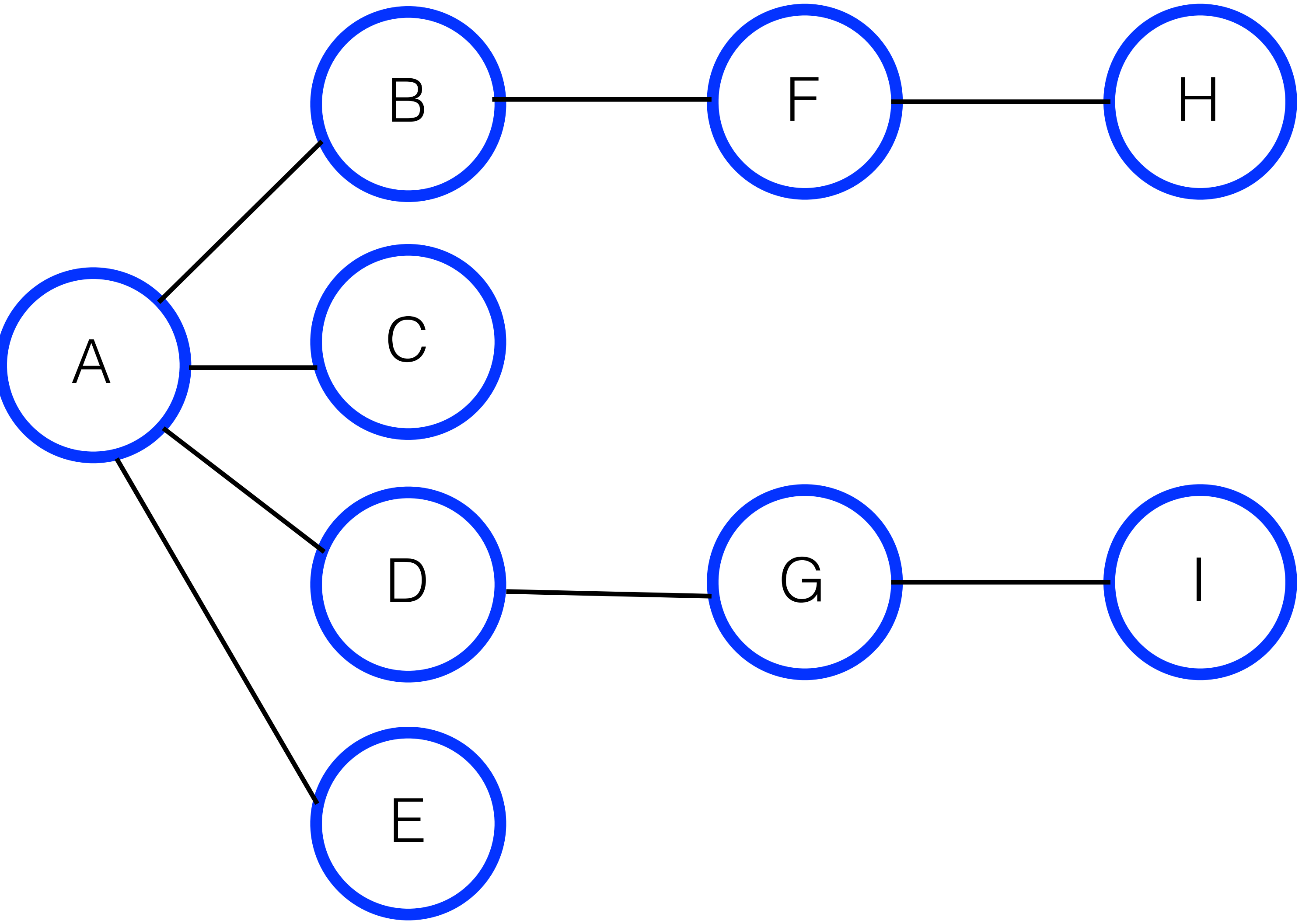



| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |
| Visit C | AC |
| Pop C | A |
| Visit D | AD |
| Visit G | ADG |
| Visit I | ADGI |
| Pop I | ADG |
| Pop G | AD |
| Pop D | A |
| Visit E | AE |




- At this point, A has no more adjacent unvisited vertices left
- We pop it off the stack

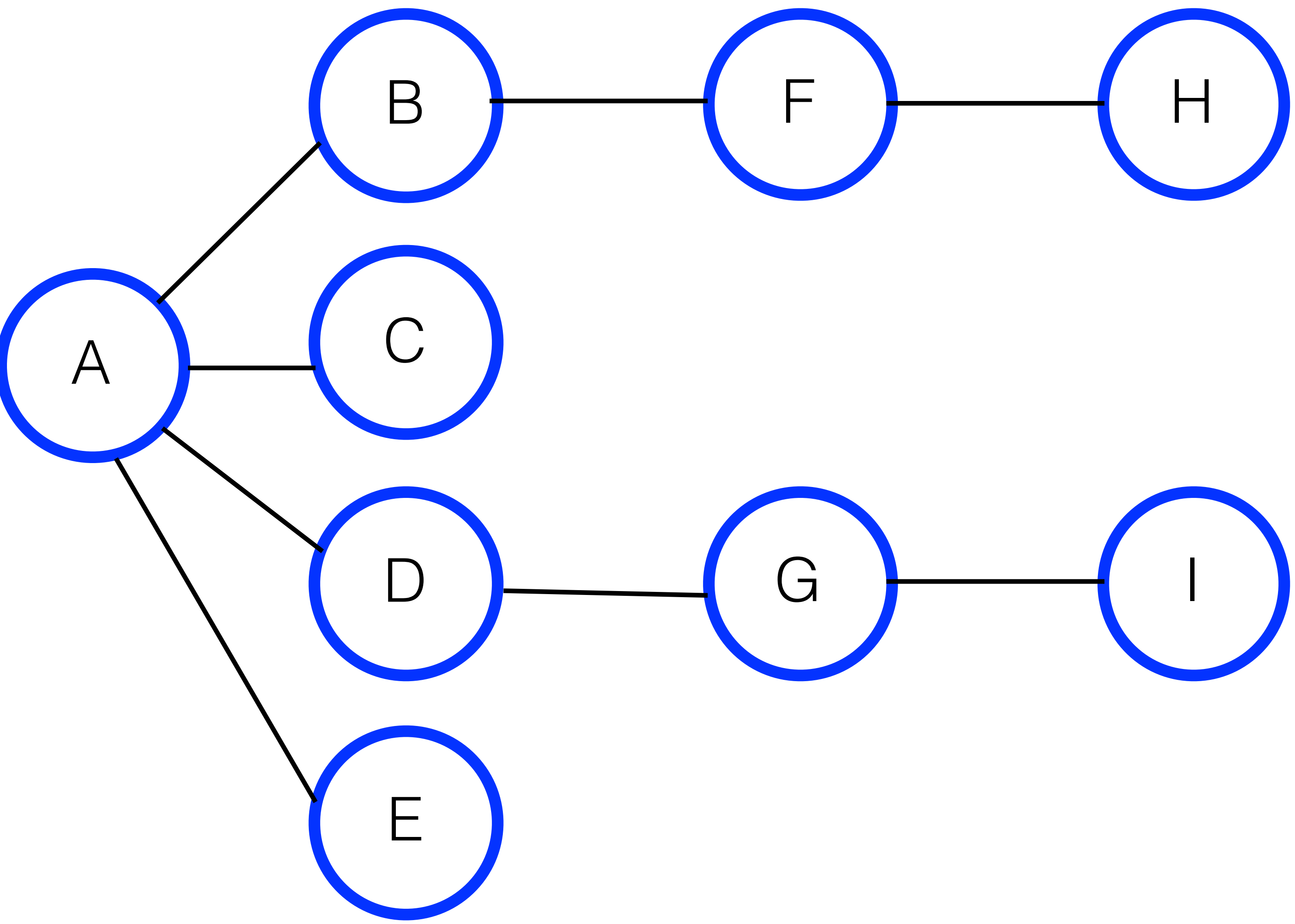
| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |
| Visit C | AC |
| Pop C | A |
| Visit D | AD |
| Visit G | ADG |
| Visit I | ADGI |
| Pop I | ADG |
| Pop G | AD |
| Pop D | A |
| Visit E | AE |
| Pop E | A |




| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |
| Visit C | AC |
| Pop C | A |
| Visit D | AD |
| Visit G | ADG |
| Visit I | ADGI |
| Pop I | ADG |
| Pop G | AD |
| Pop D | A |
| Visit E | AE |
| Pop E | A |
| Pop A | |
| Done | |

- This brings us to **Rule 3:**

“If you cannot follow Rule 1 or Rule 2, you are done”




**Order:** ABFHCDGIE

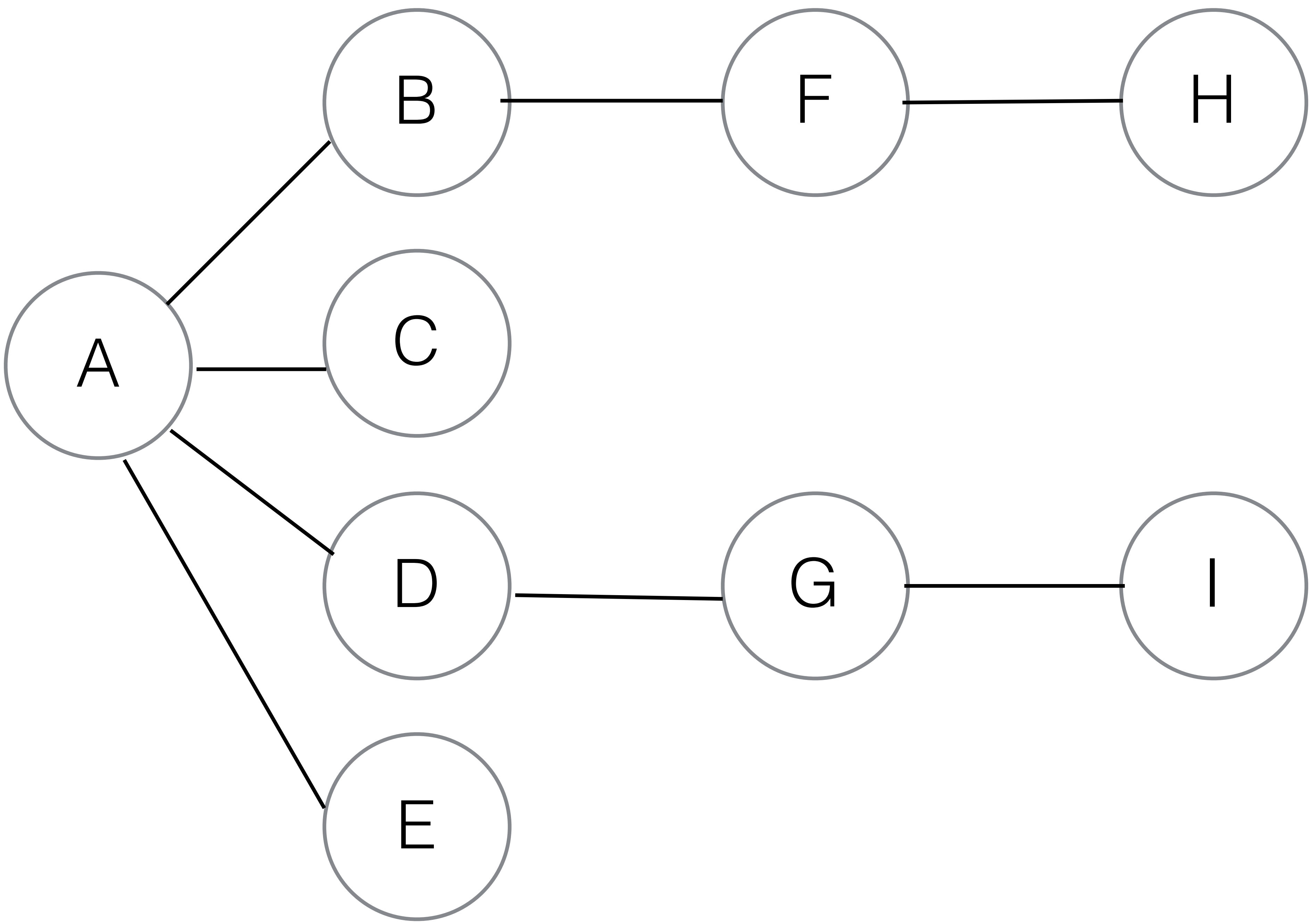
**Time:** $O(|V| + |E|)$

| Event | Stack |
|---|---|
| **Visit A** | A |
| **Visit B** | AB |
| **Visit F** | ABF |
| **Visit H** | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |
| **Visit C** | AC |
| Pop C | A |
| **Visit D** | AD |
| **Visit G** | ADG |
| **Visit I** | ADGI |
| Pop I | ADG |
| Pop G | AD |
| Pop D | A |
| **Visit E** | AE |
| Pop E | A |
| Pop A | |
| Done | |

# DFS

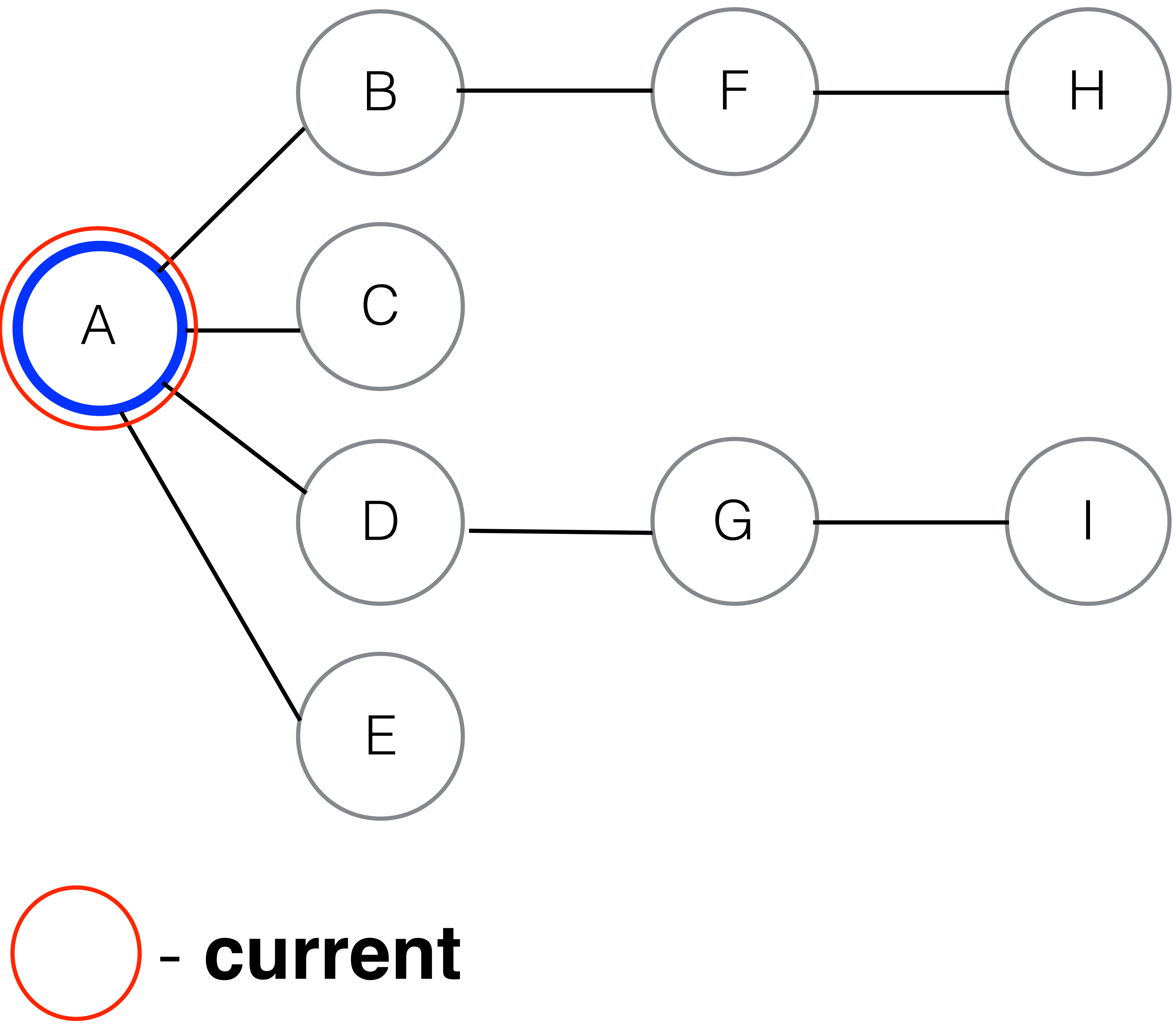
- Notice that,
    - DFS tries to get as far away from the starting point as quickly as possible
    - And returns only when it reaches a dead end
    - Thus the name, **Depth First Search**

# BFS with a Queue

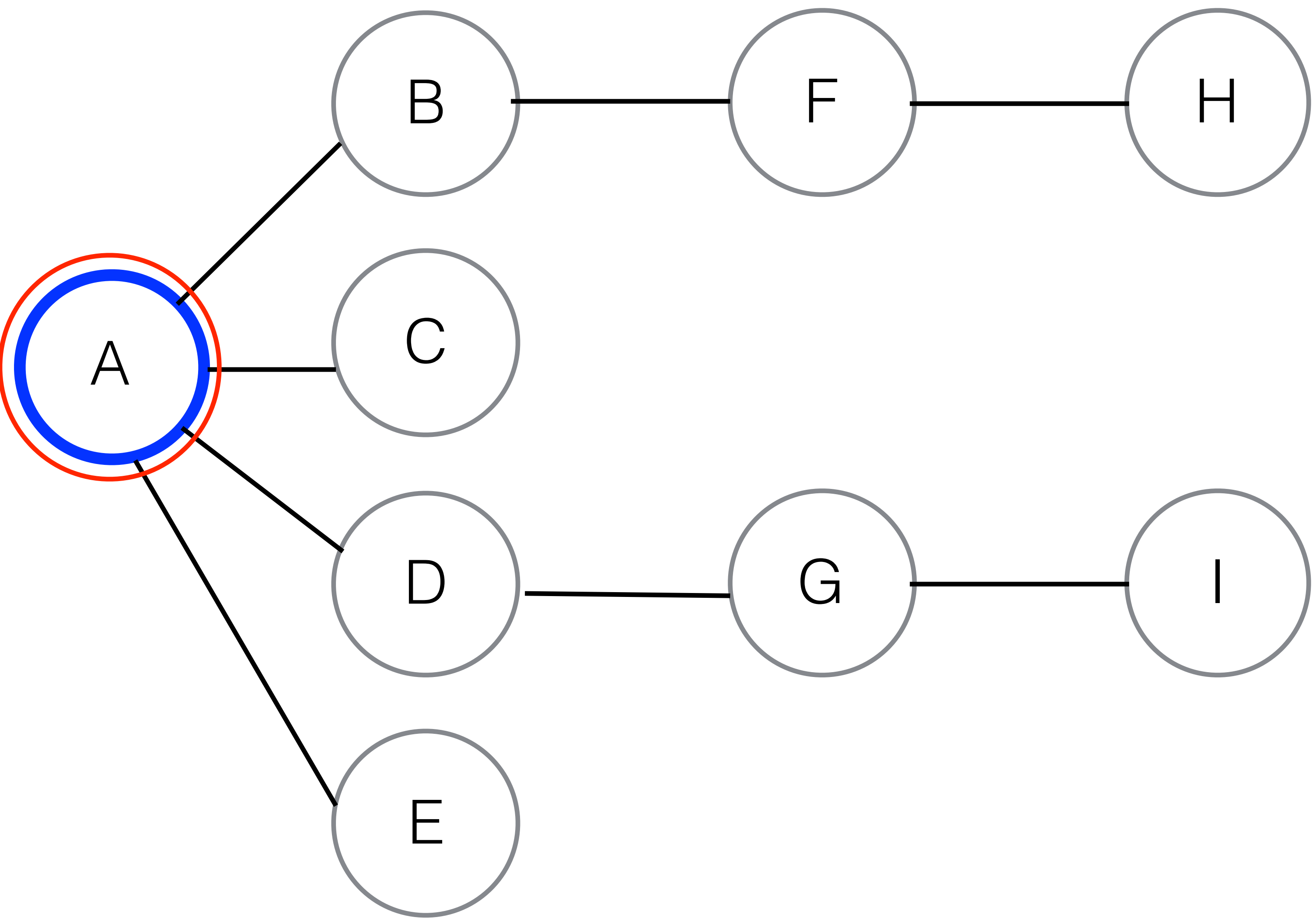


# BFS with a Queue (2)

- Start with a vertex, visit it, and call it **current**
- Let's start with vertex A




| Event | Queue |
| --- | --- |
| Visit A | |



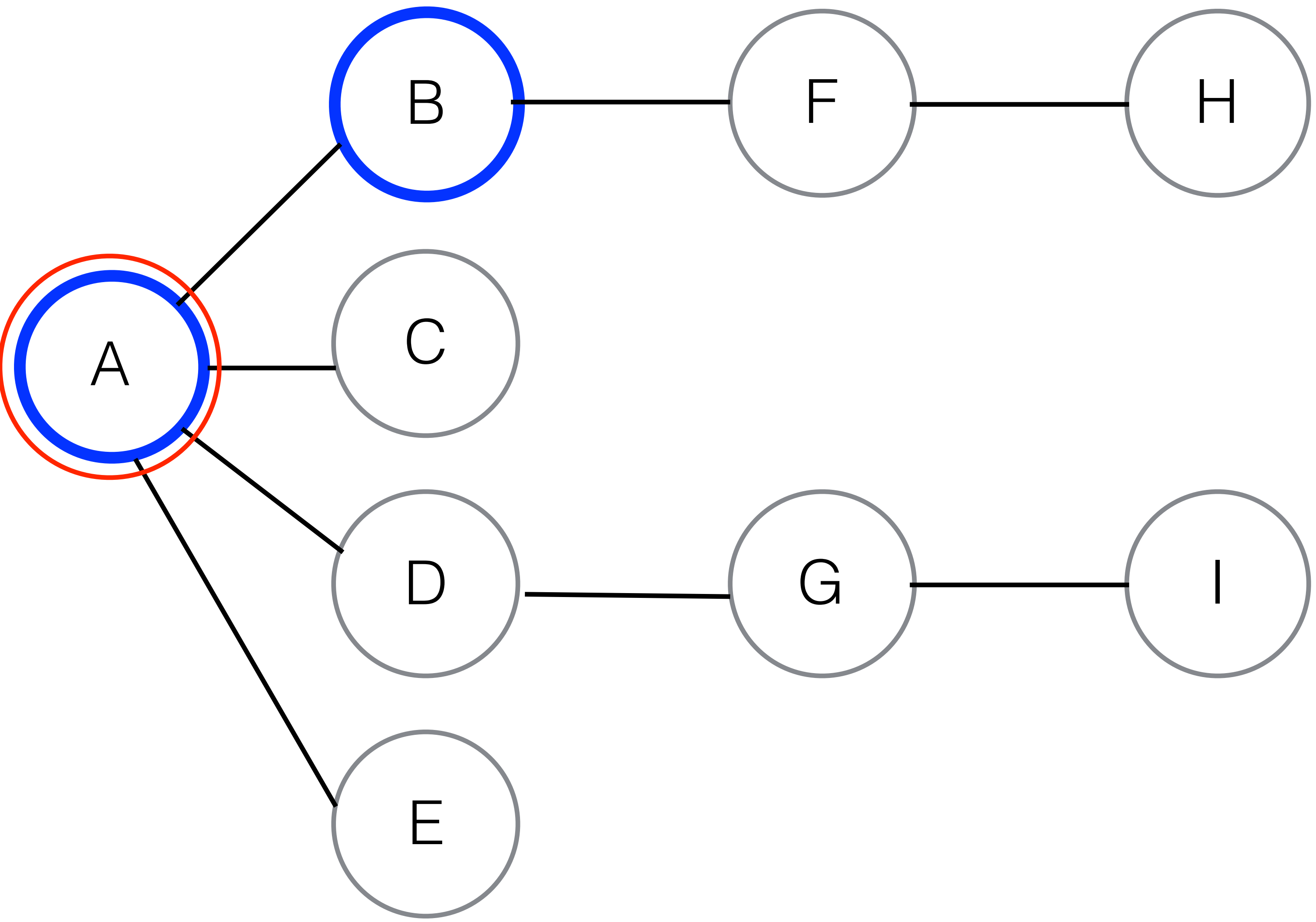

| Event | Queue |
|---|---|
| Visit A | |

- **current**

Notice that the **current** is not inserted into the queue

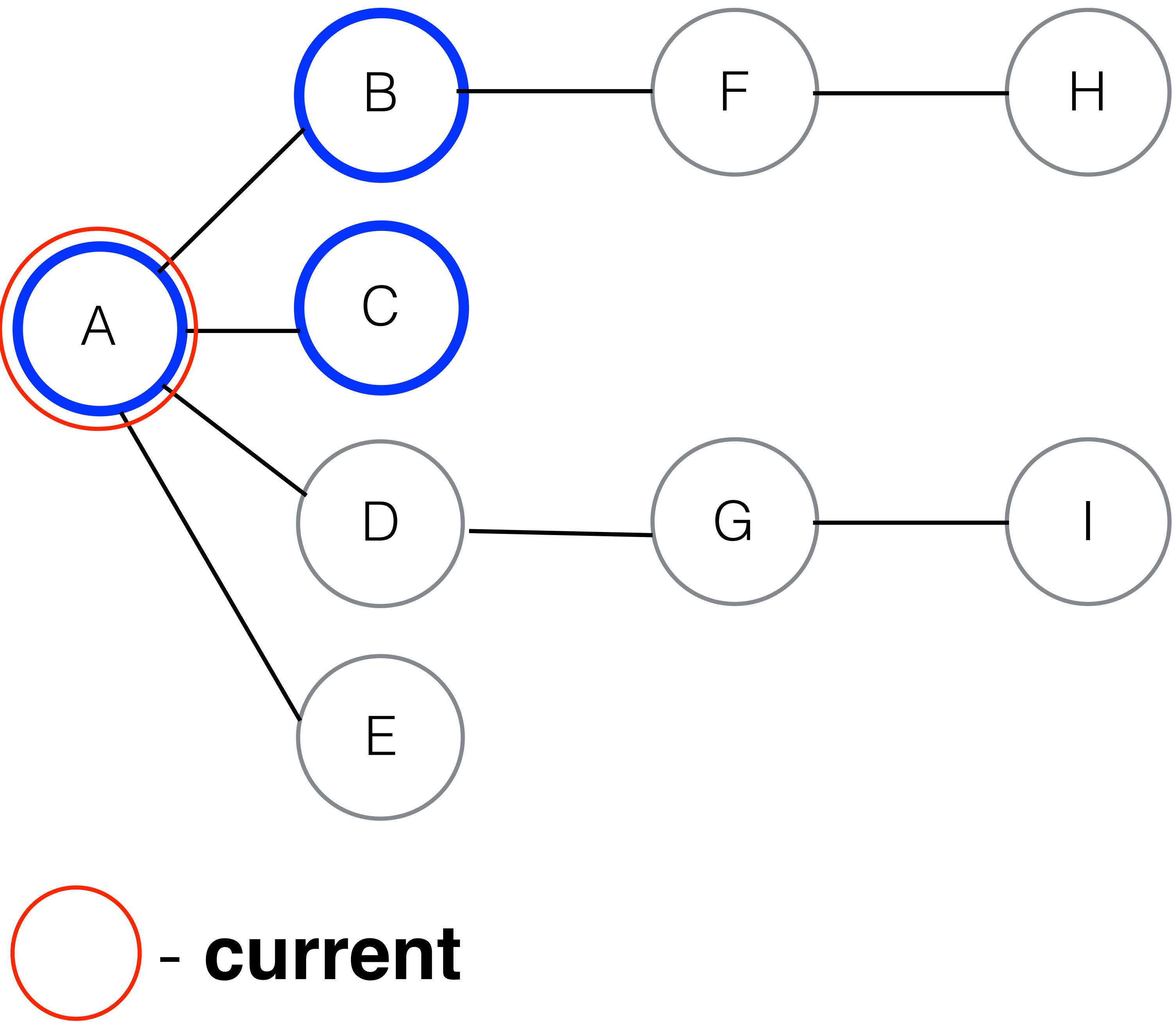
Now follow this rule

**Rule 1:** Visit the next unvisited vertex (if there is one) that is adjacent to the **current** vertex, mark it, and insert it into the queue

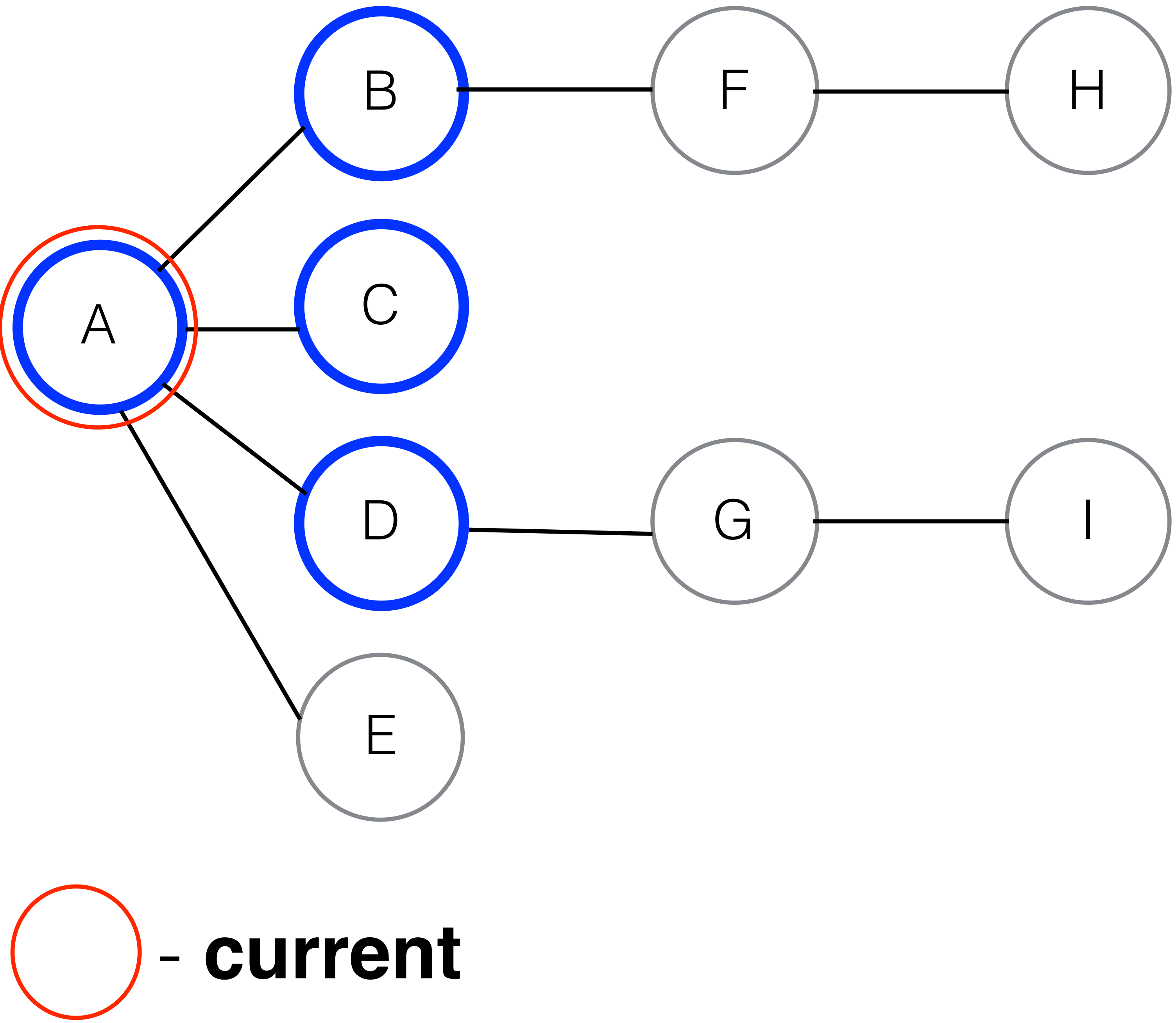



| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |

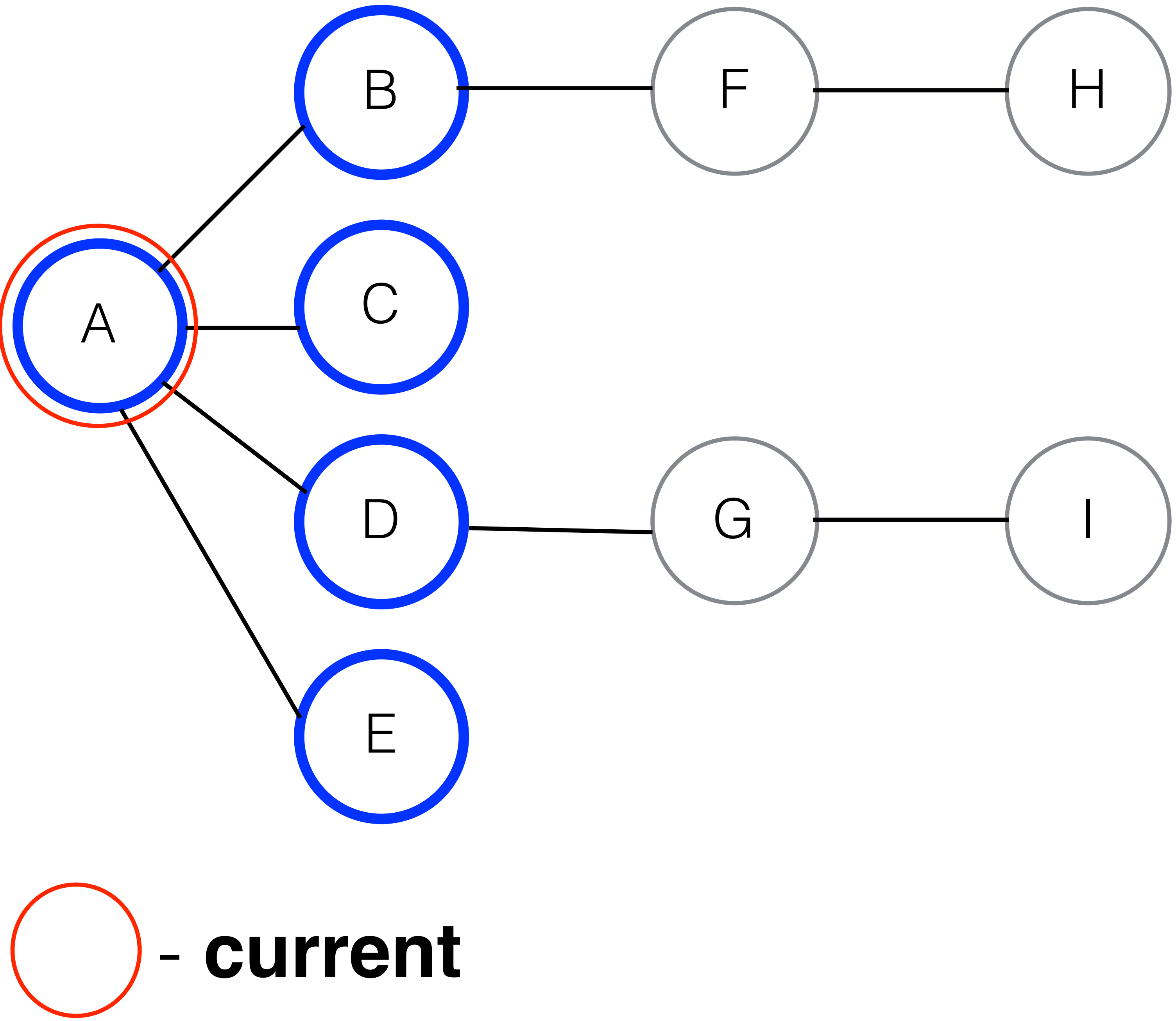
- **current**



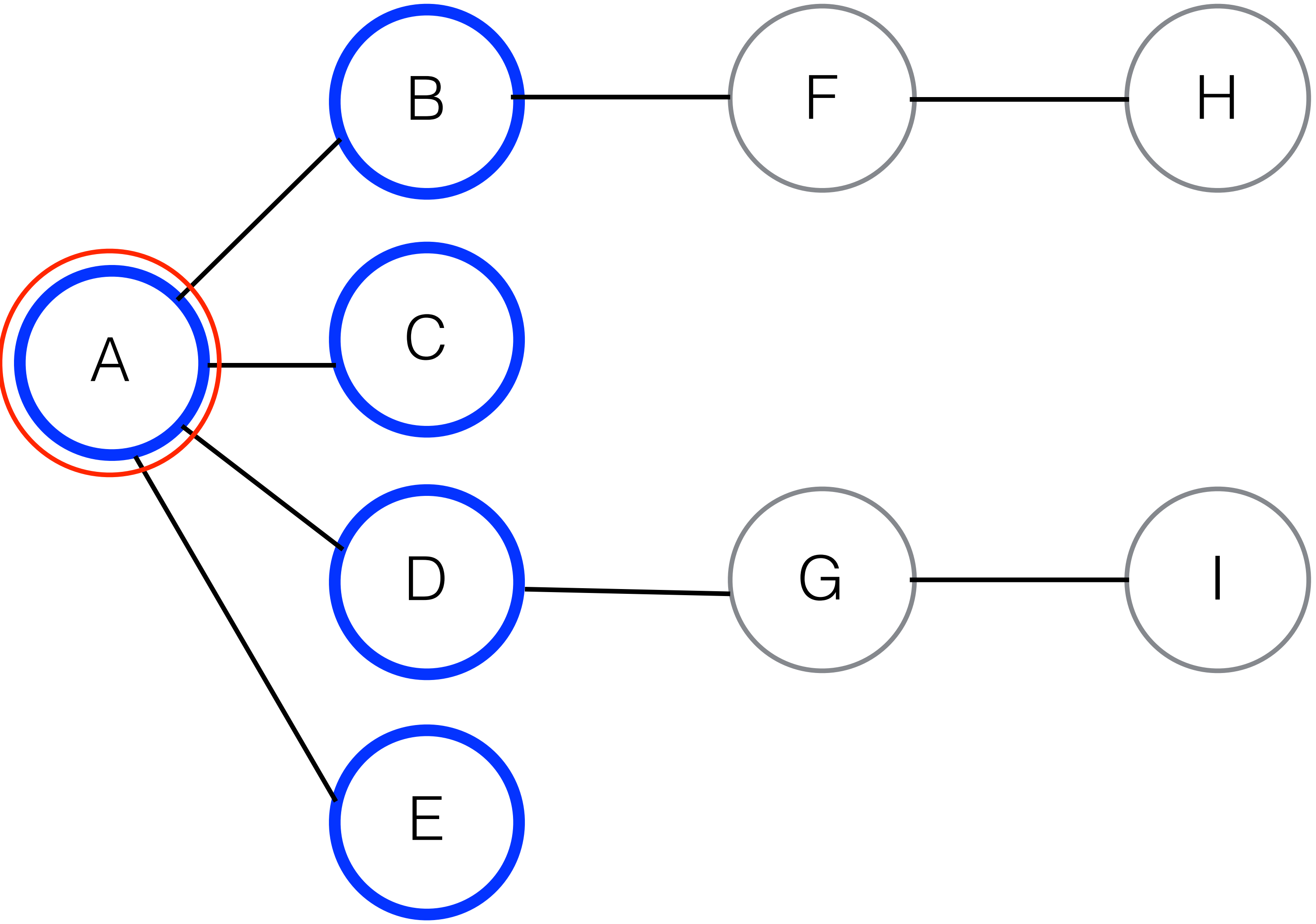

| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |



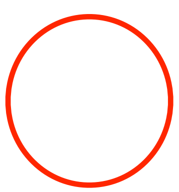

| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |




| Event | Queue |
| --- | --- |
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |



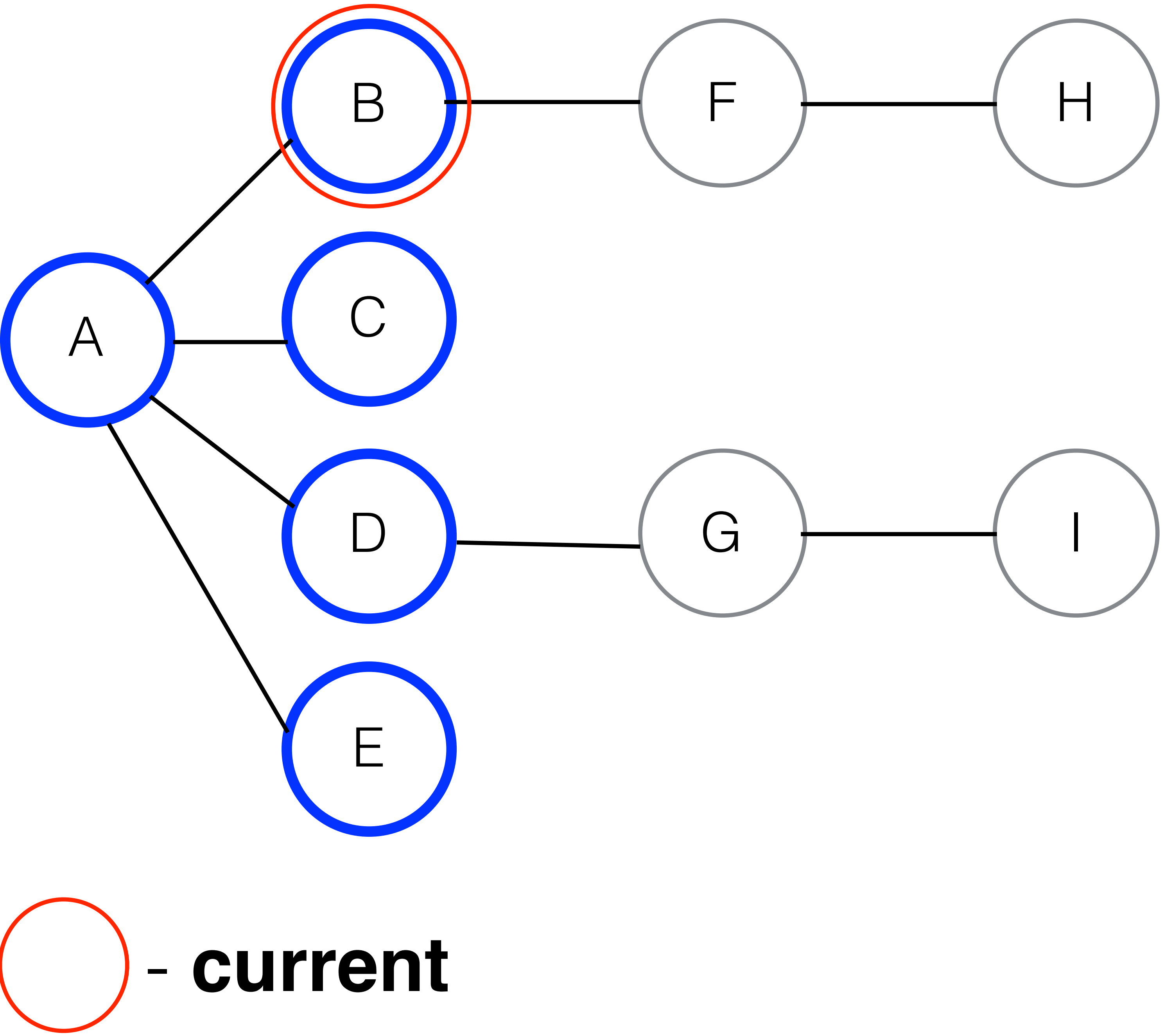

| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |



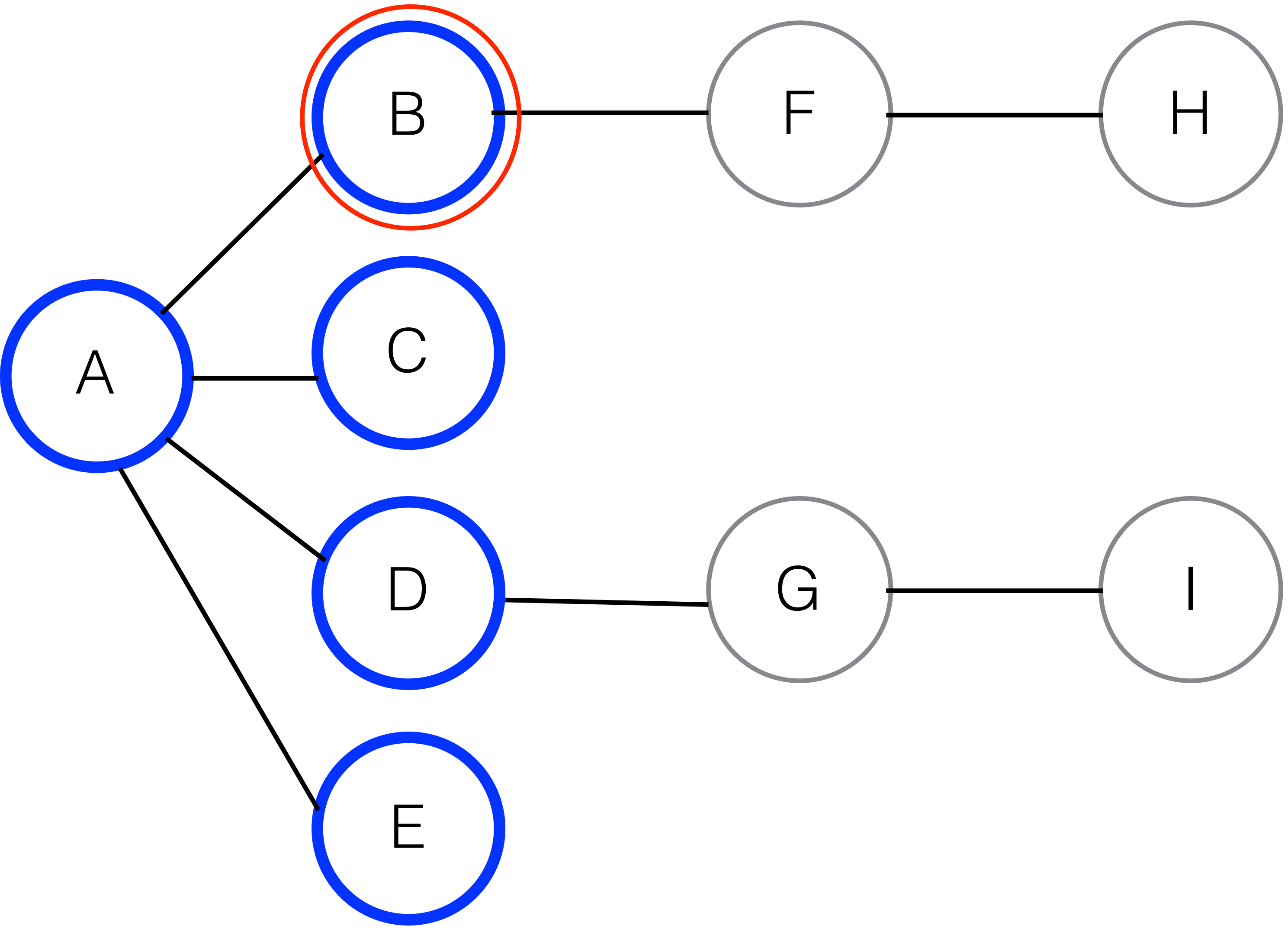
- **current**

At this point A (**the current**) has no more unvisited adjacent vertex
So, follow **Rule 2**:

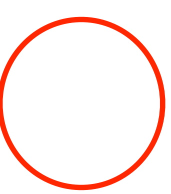
If you can't carry out Rule 1 because there are no more unvisited vertices, remove a vertex from the queue (if possible) and make it **current** vertex



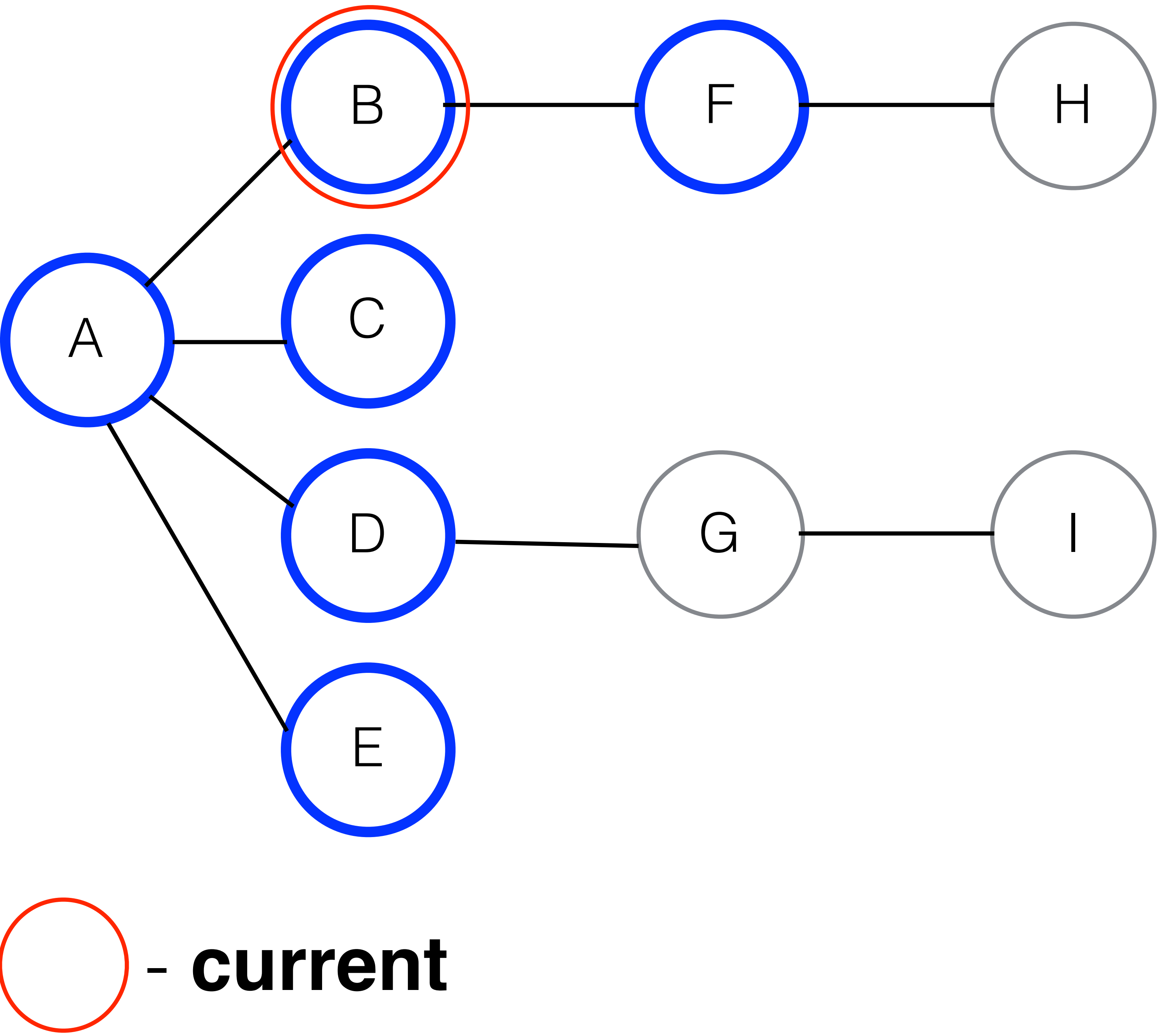

| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |




| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |

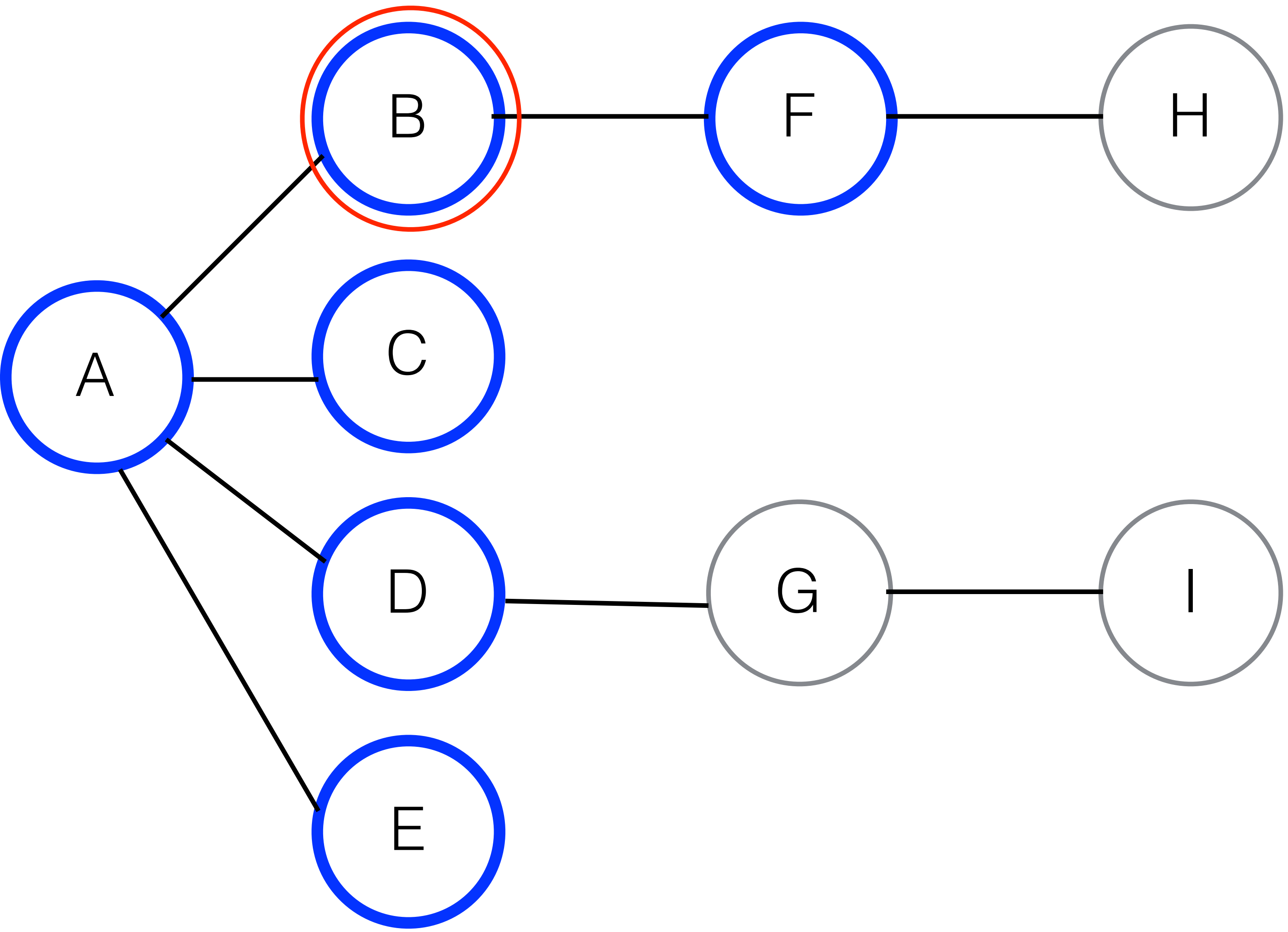
 - **current**

Repeat Rule 1 for the new **current**

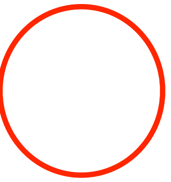



| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |

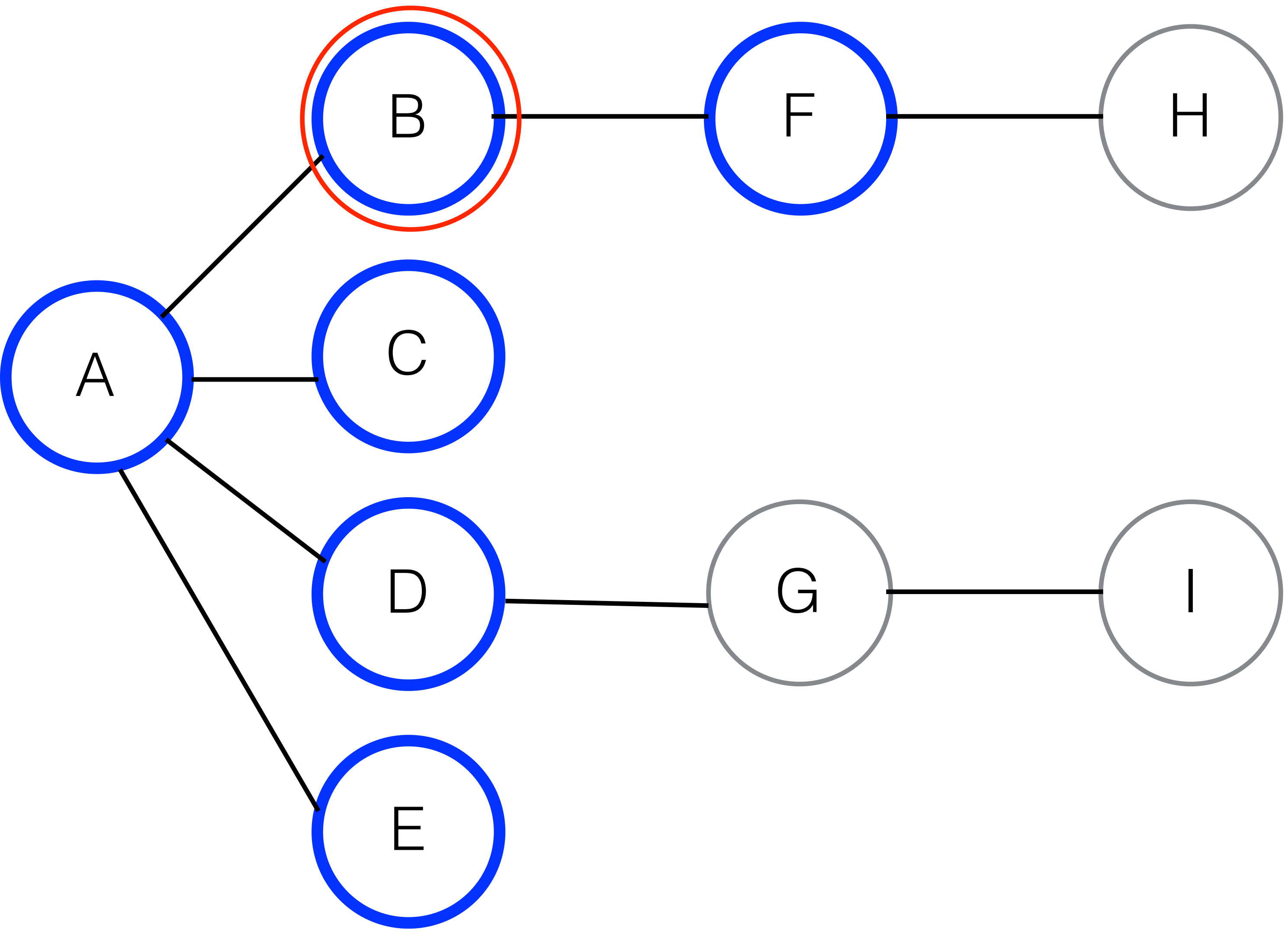
Will we follow BA?



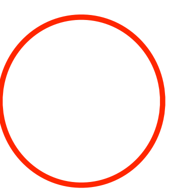

| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |

 - **current**

Will we follow BA?
Yes! But it will take us back to A, which is already visited!




| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |

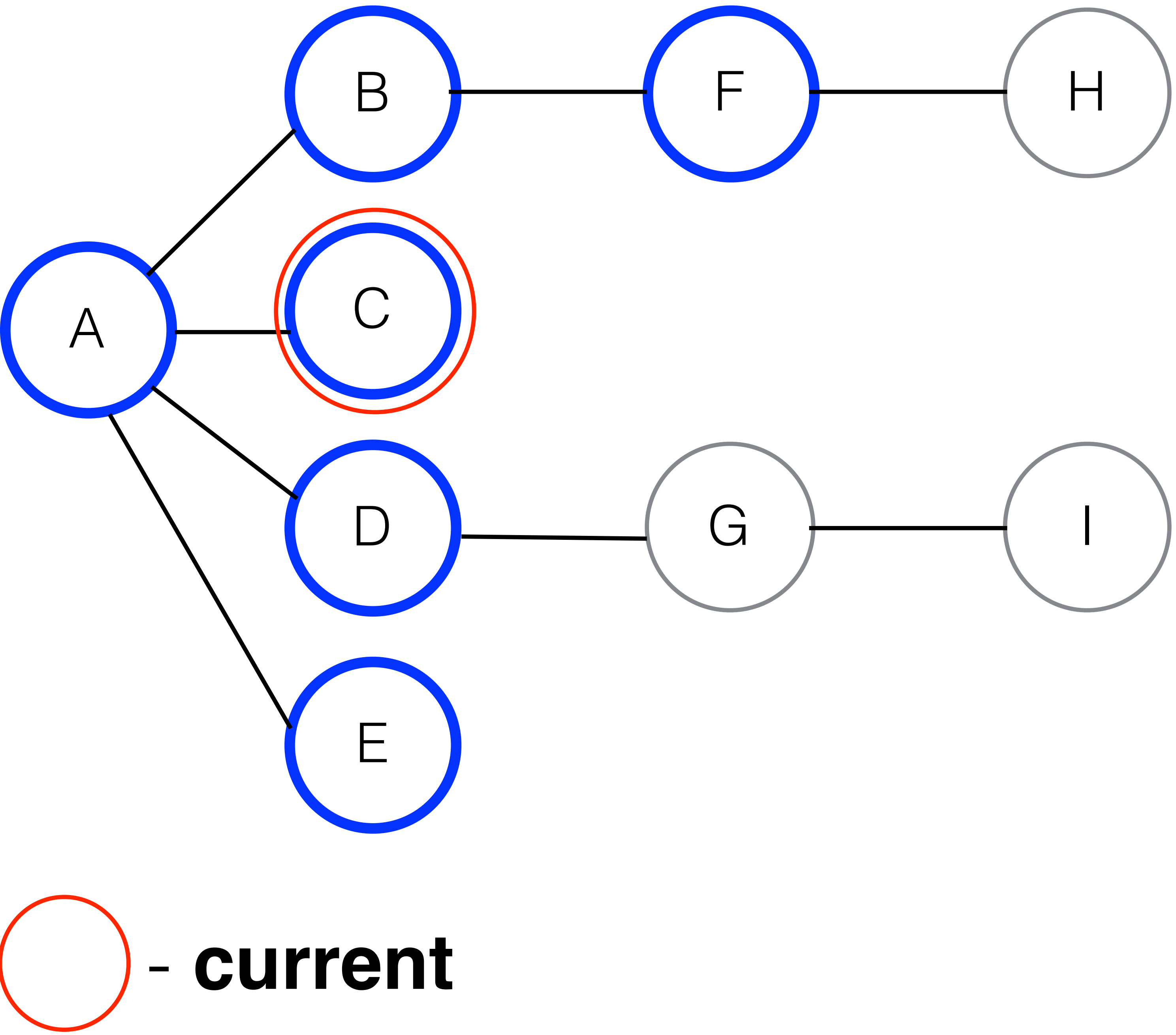


- **current**

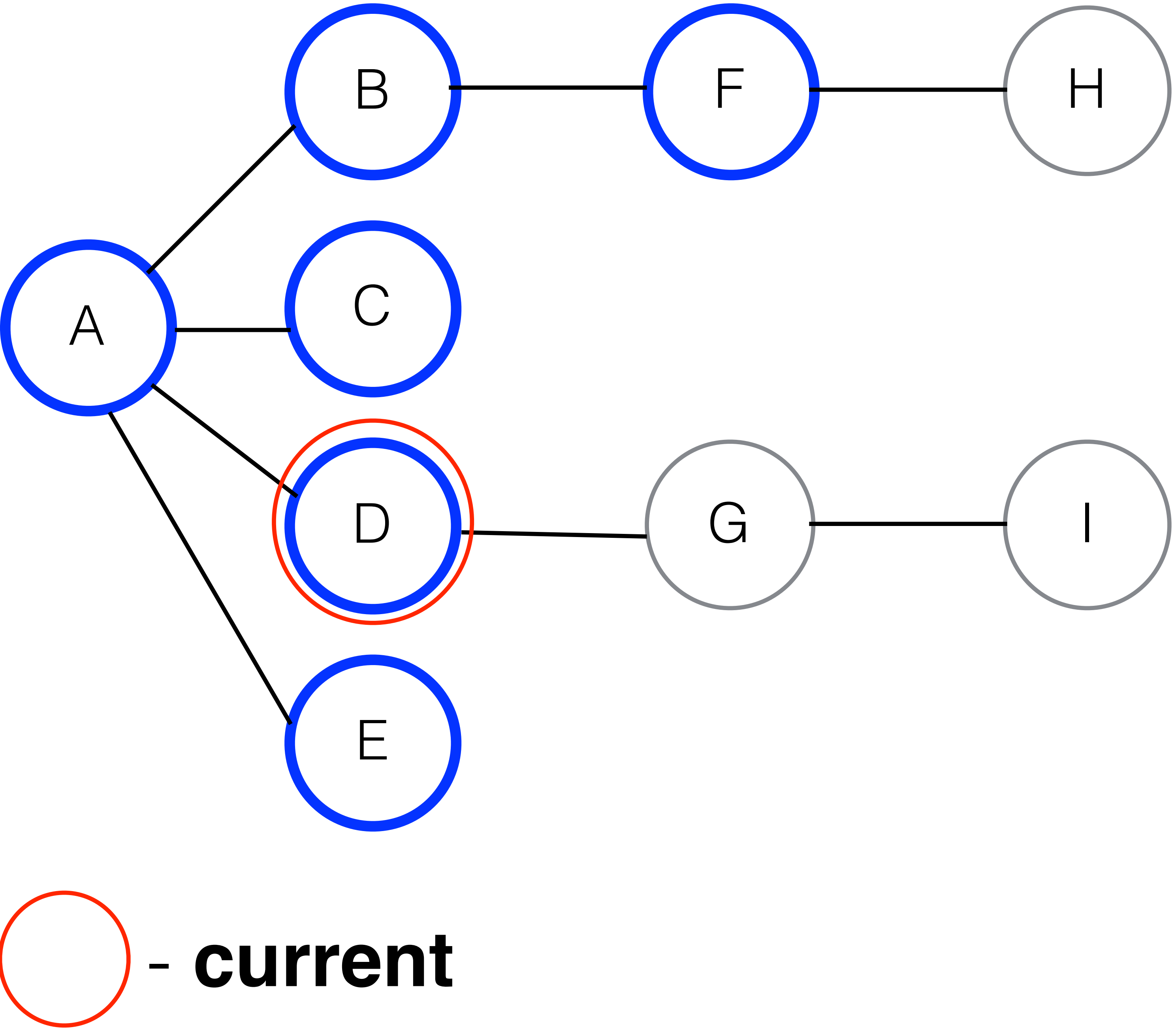
Will we follow BA?

Yes! But it will take us back to A, which is already visited!

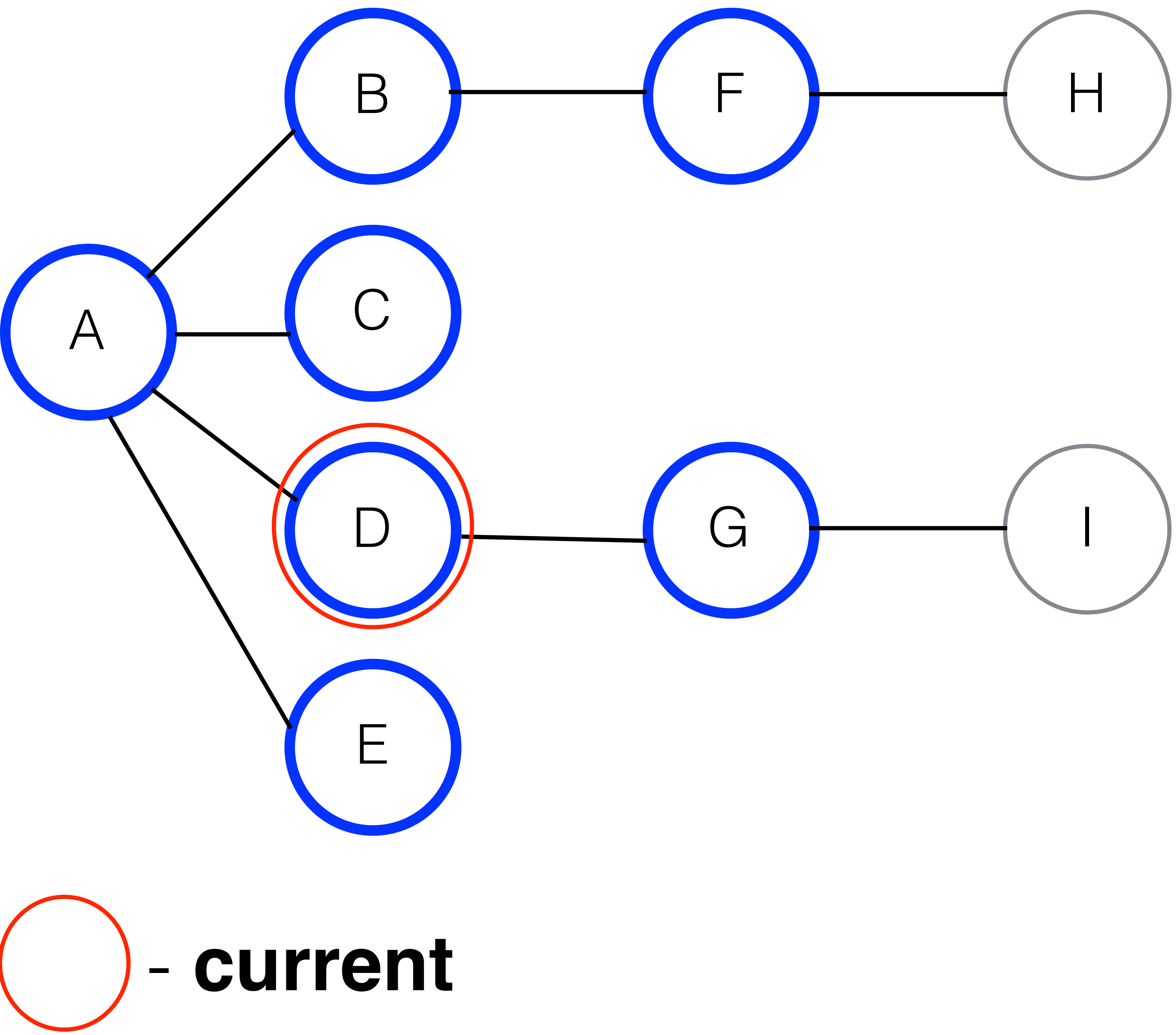
Thus each vertex is visited once, and each edge twice!



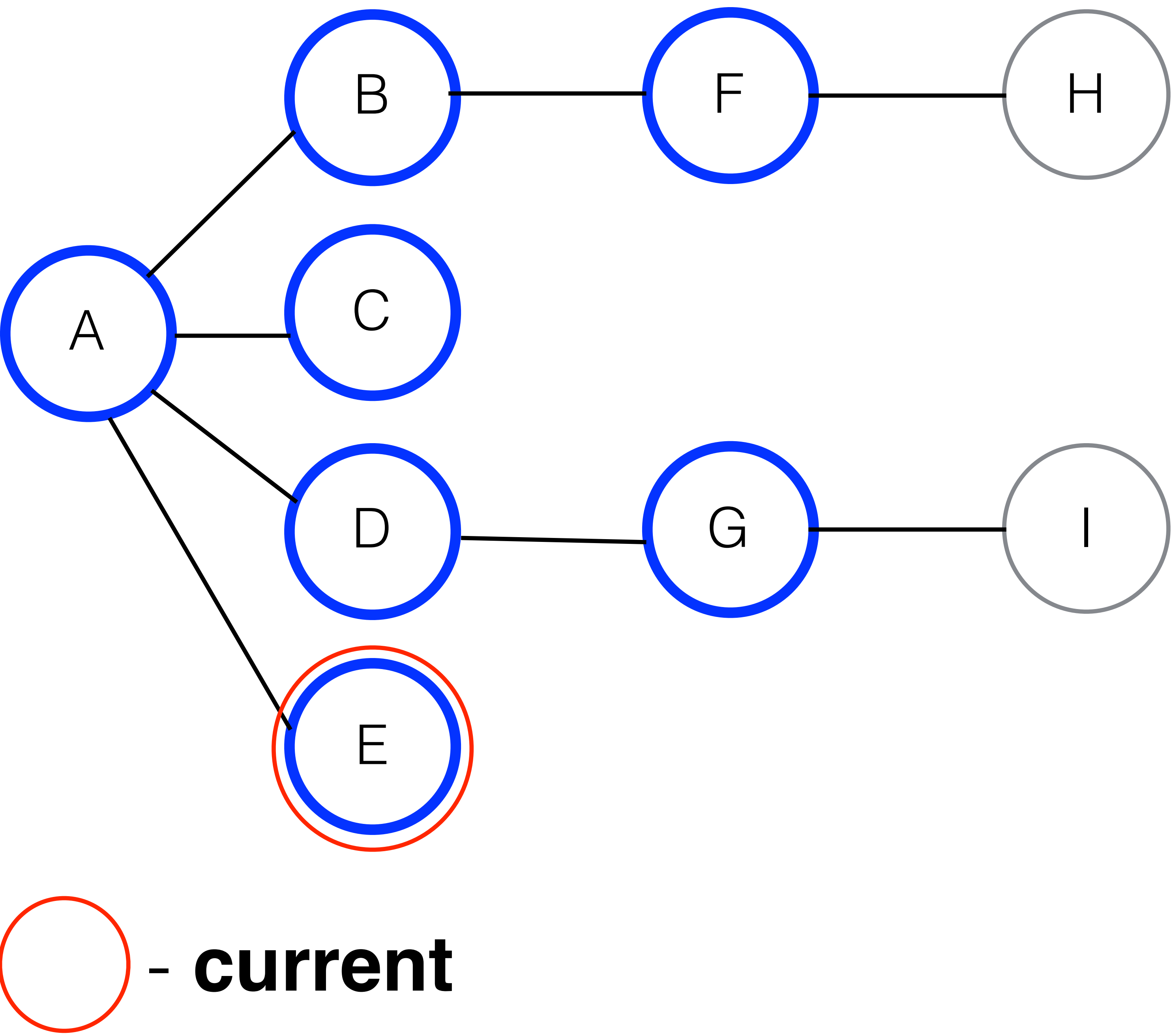

| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |
| Dequeue (C) | DEF |



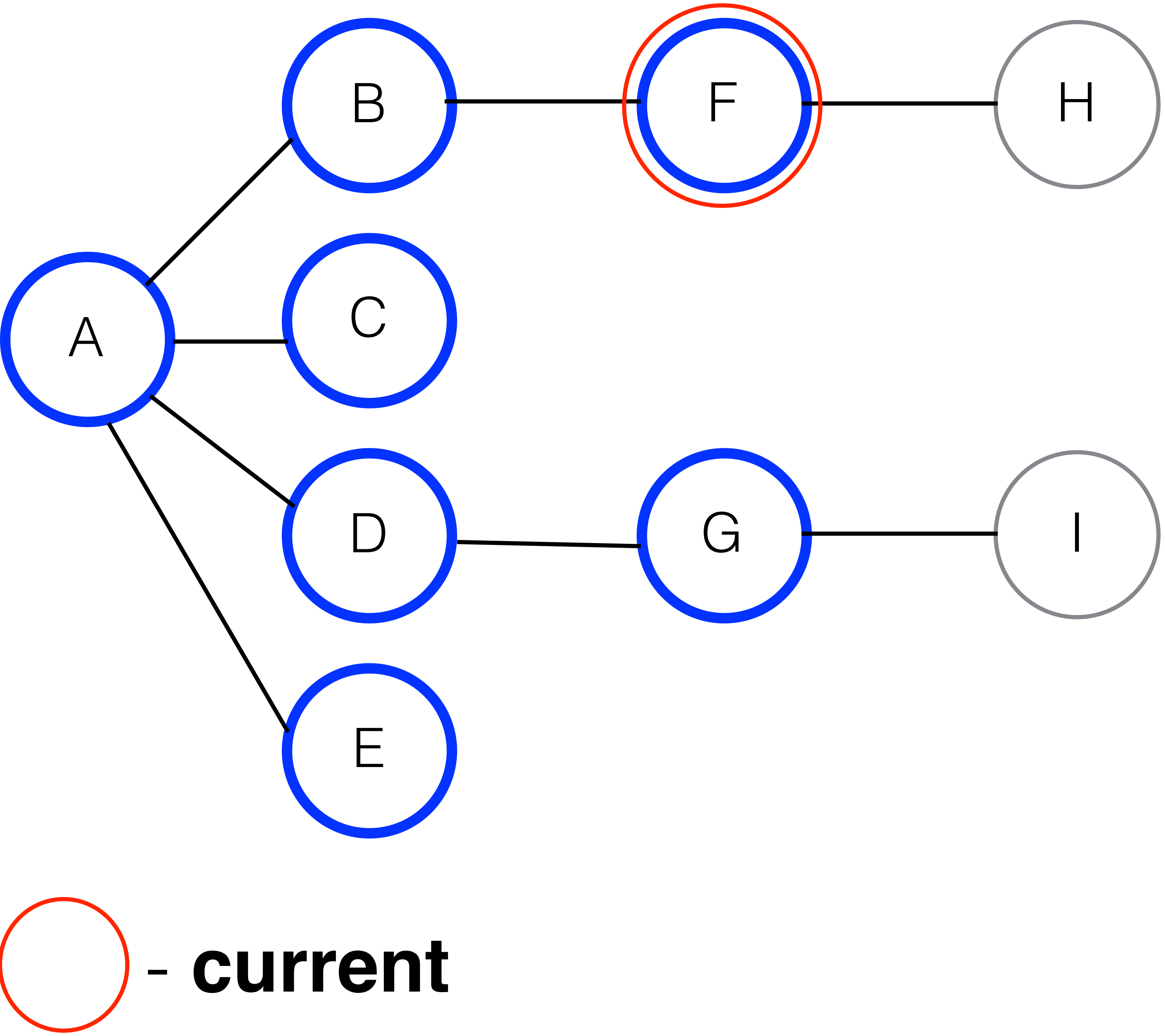

| Event | Queue |
| --- | --- |
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |
| Dequeue (C) | DEF |
| Dequeue (D) | EF |



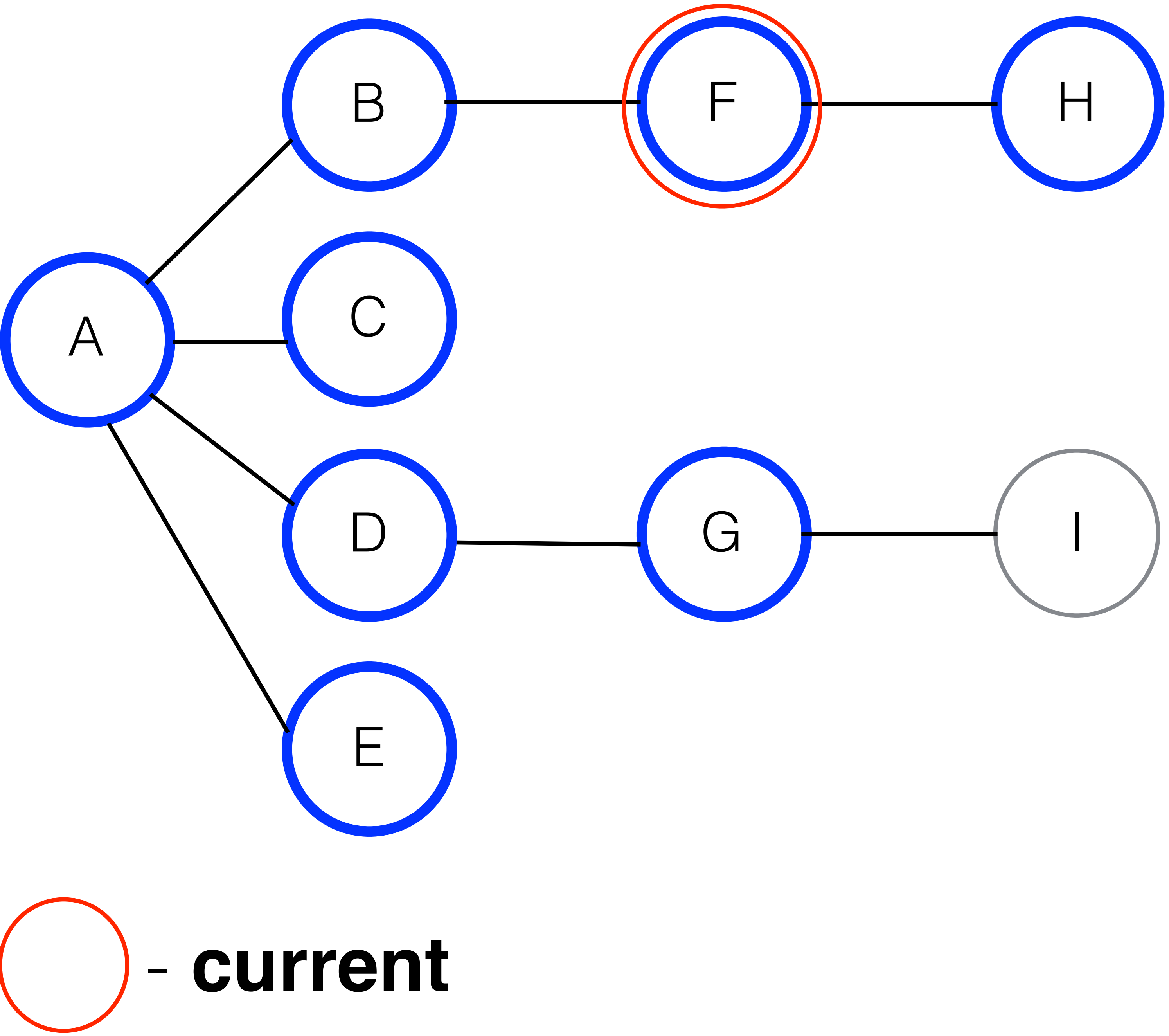

| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |
| Dequeue (C) | DEF |
| Dequeue (D) | EF |
| Visit G | EFG |



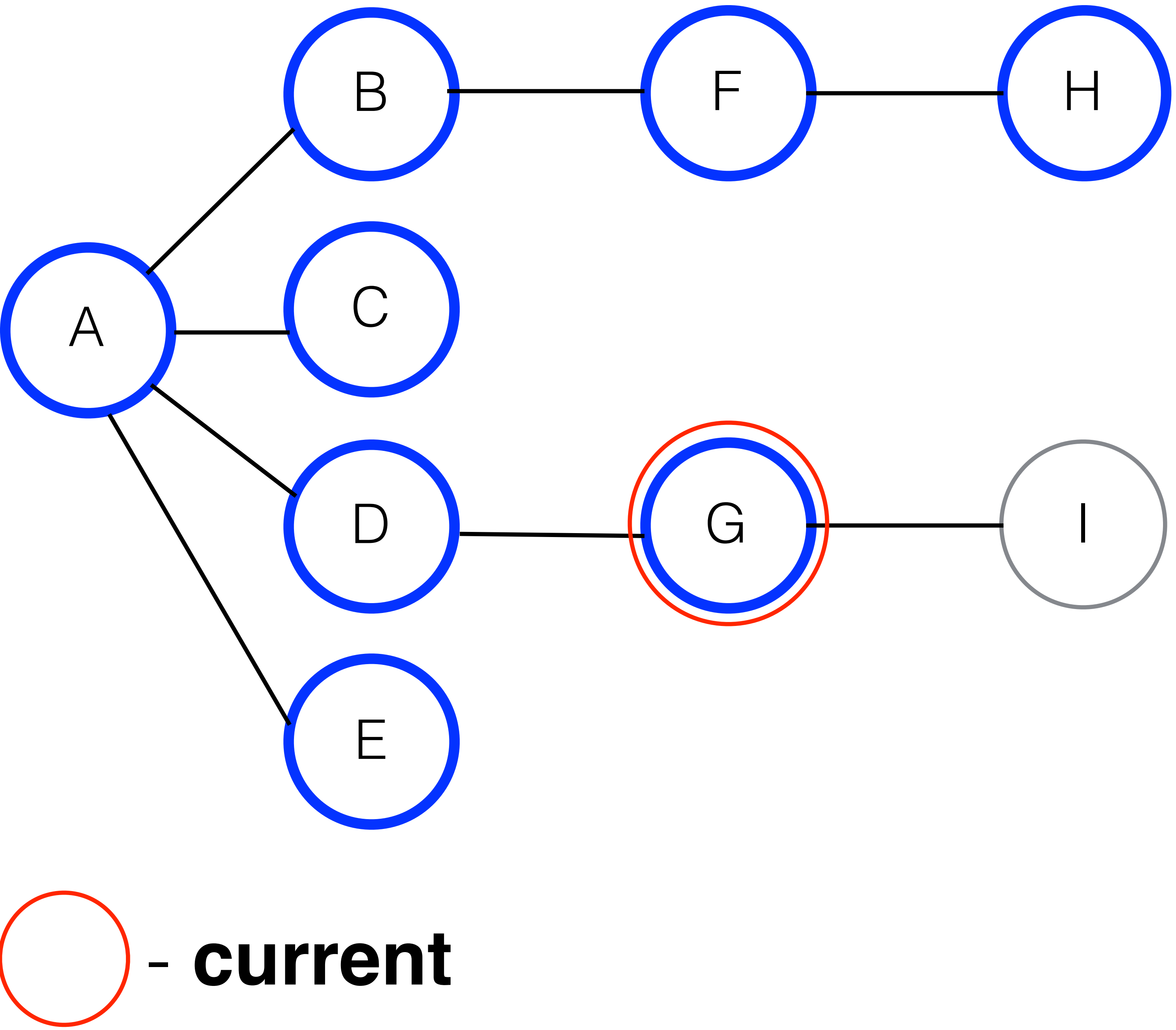

| Event | Queue |
| --- | --- |
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |
| Dequeue (C) | DEF |
| Dequeue (D) | EF |
| Visit G | EFG |
| Dequeue (E) | FG |



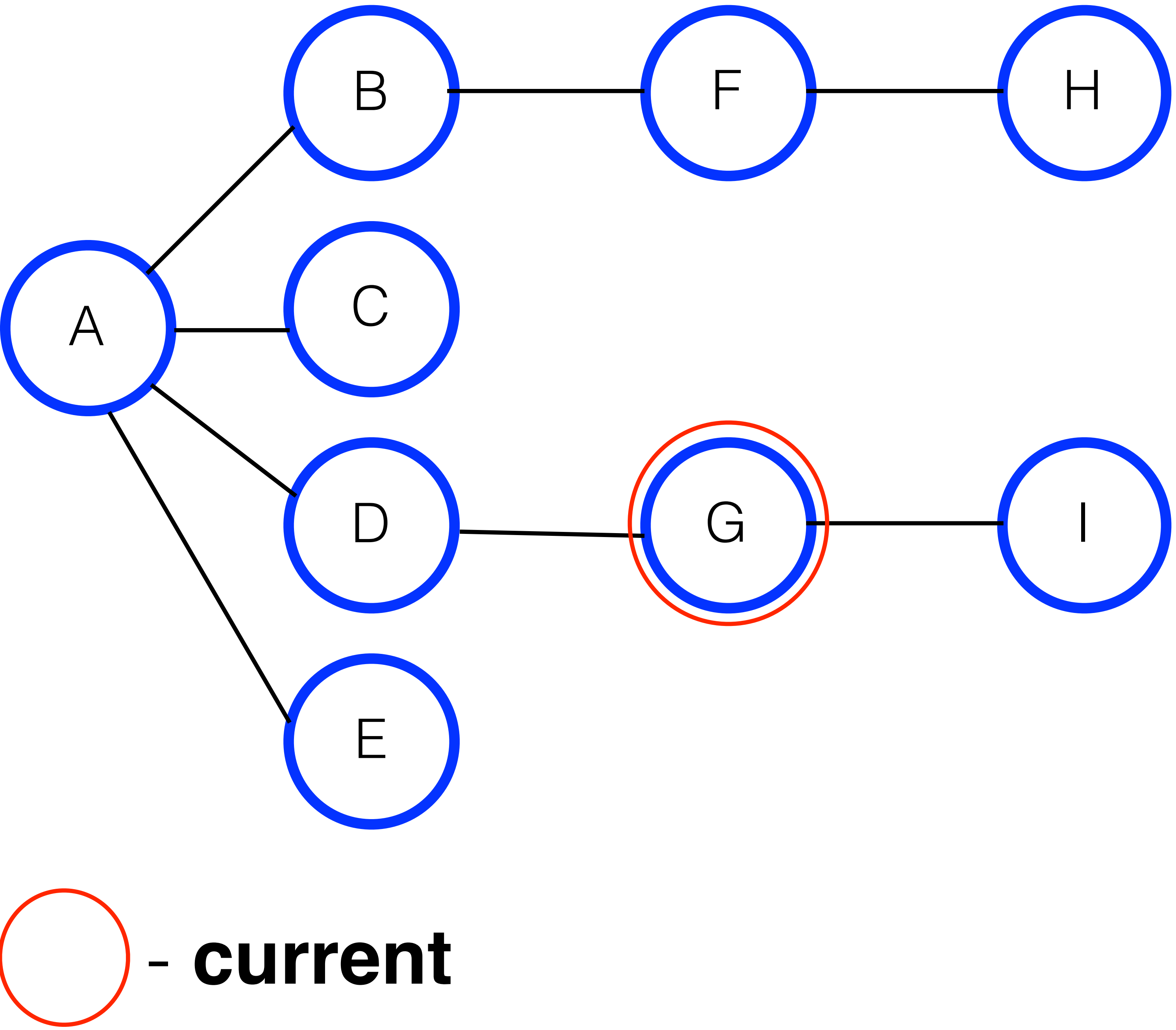

| Event | Queue |
| --- | --- |
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |
| Dequeue (C) | DEF |
| Dequeue (D) | EF |
| Visit G | EFG |
| Dequeue (E) | FG |
| Dequeue (F) | G |



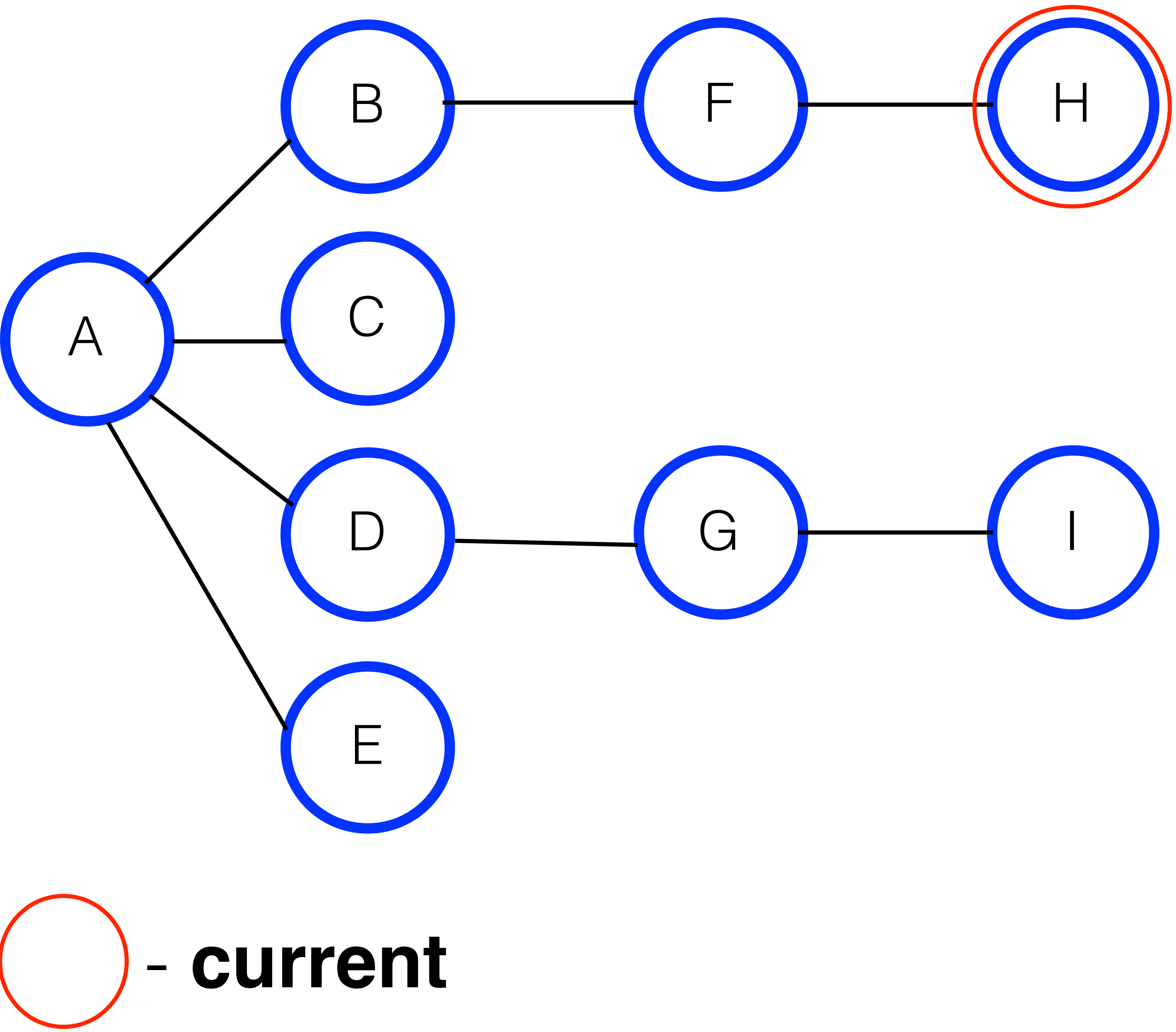

| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |
| Dequeue (C) | DEF |
| Dequeue (D) | EF |
| Visit G | EFG |
| Dequeue (E) | FG |
| Dequeue (F) | G |
| Visit H | GH |



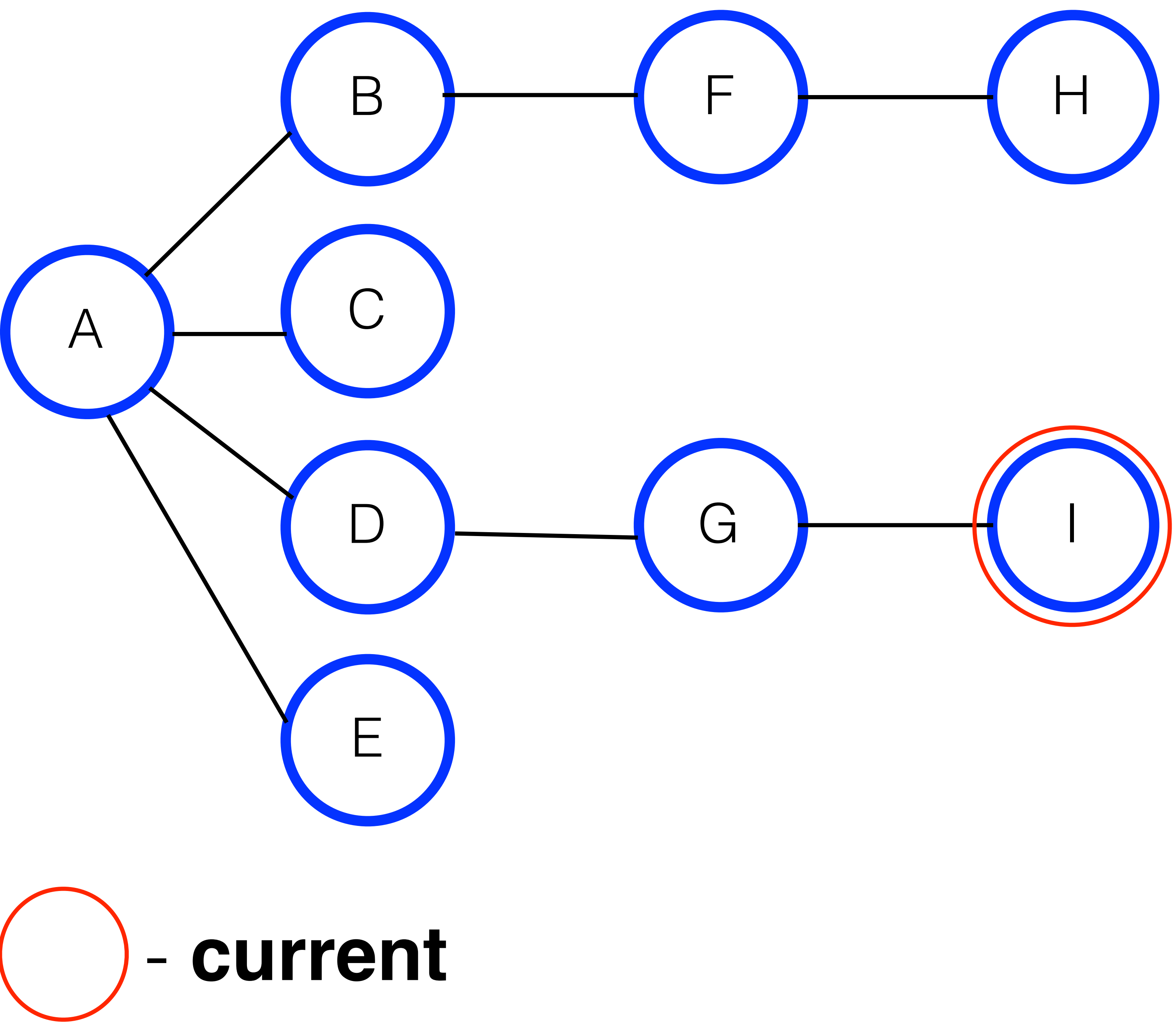

| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |
| Dequeue (C) | DEF |
| Dequeue (D) | EF |
| Visit G | EFG |
| Dequeue (E) | FG |
| Dequeue (F) | G |
| Visit H | GH |
| Dequeue (G) | H |



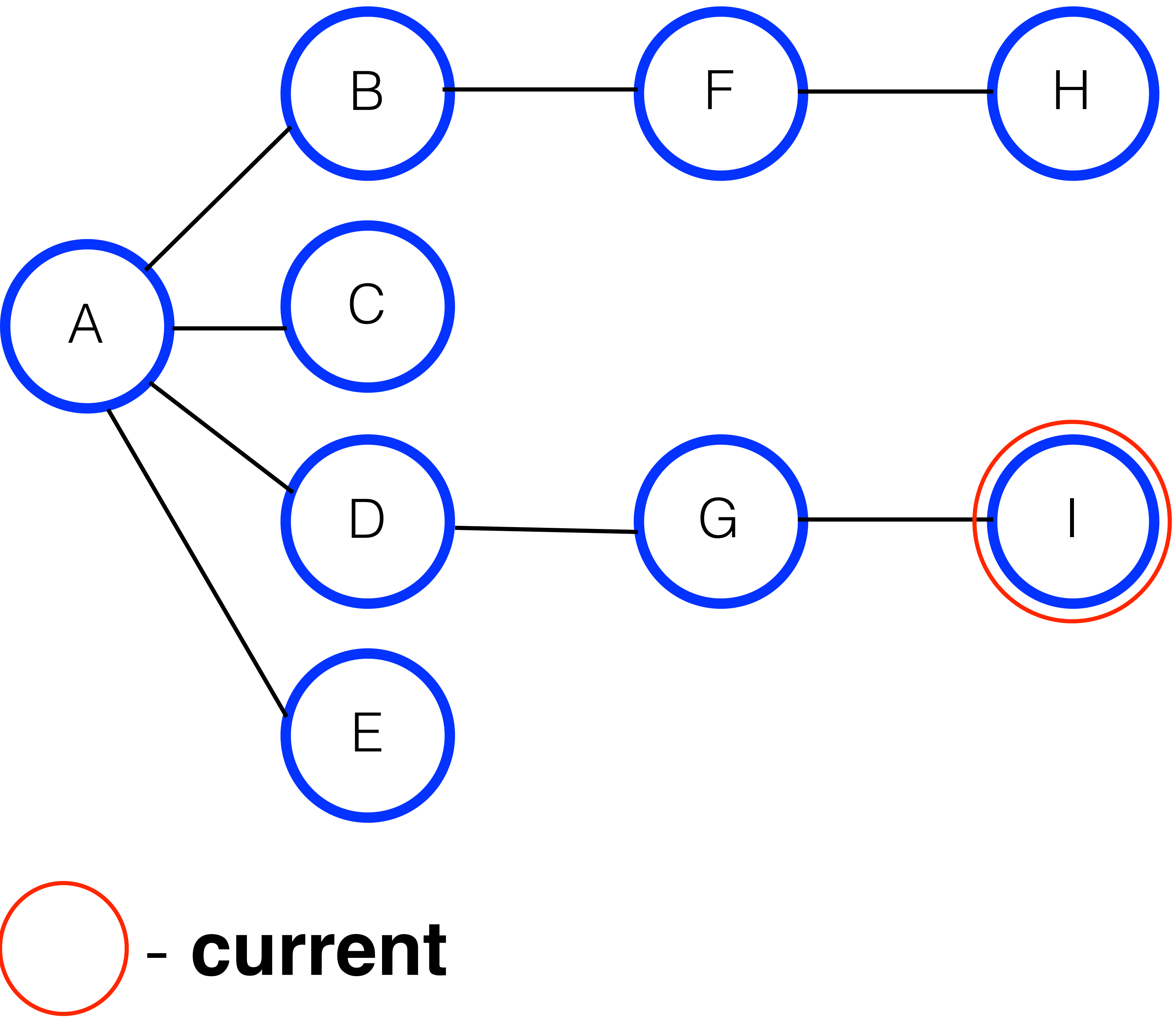

| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |
| Dequeue (C) | DEF |
| Dequeue (D) | EF |
| Visit G | EFG |
| Dequeue (E) | FG |
| Dequeue (F) | G |
| Visit H | GH |
| Dequeue (G) | H |
| Visit I | HI |




| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |
| Dequeue (C) | DEF |
| Dequeue (D) | EF |
| Visit G | EFG |
| Dequeue (E) | FG |
| Dequeue (F) | G |
| Visit H | GH |
| Dequeue (G) | H |
| Visit I | HI |
| Dequeue (H) | I |

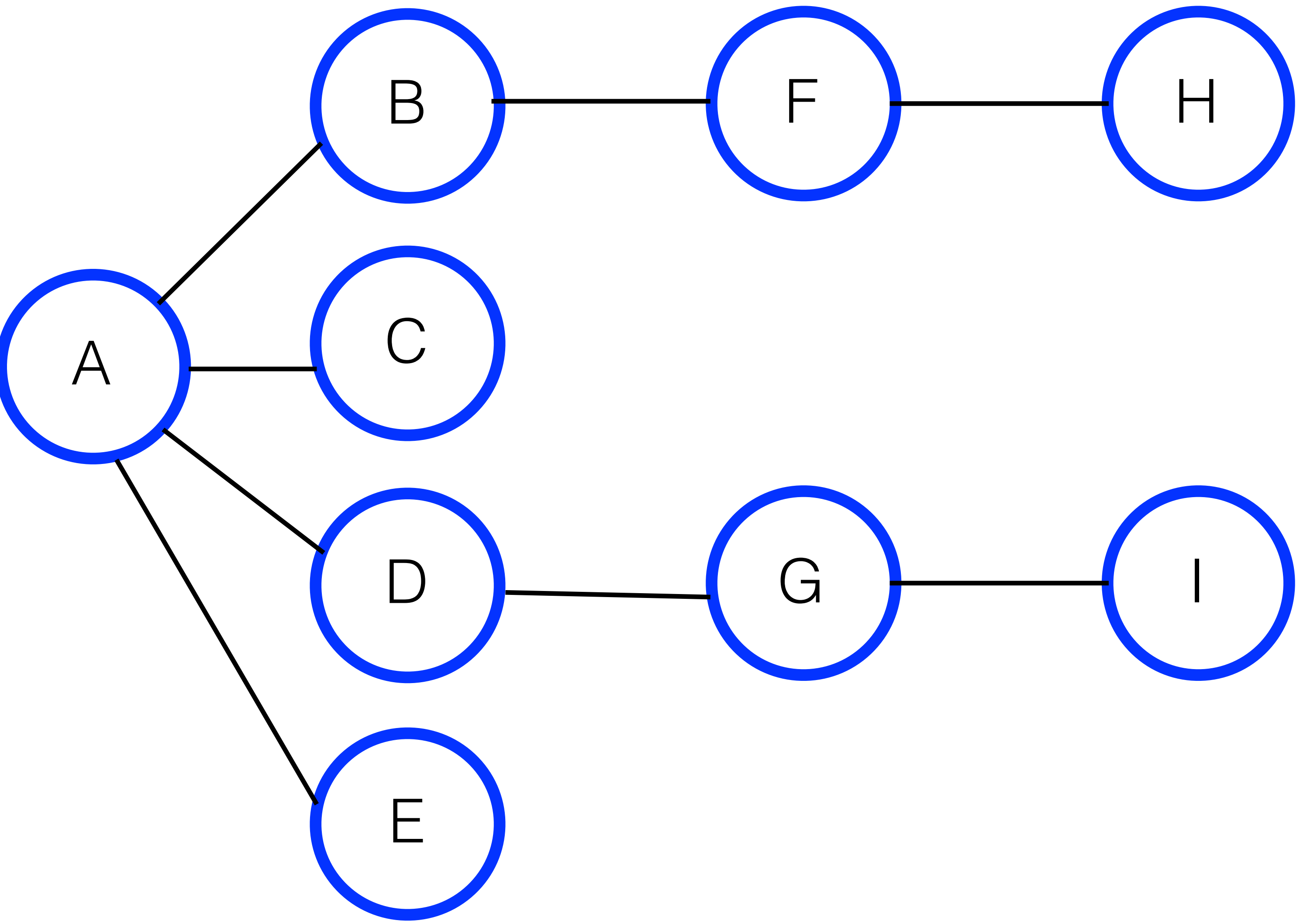



| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |
| Dequeue (C) | DEF |
| Dequeue (D) | EF |
| Visit G | EFG |
| Dequeue (E) | FG |
| Dequeue (F) | G |
| Visit H | GH |
| Dequeue (G) | H |
| Visit I | HI |
| Dequeue (H) | I |
| Dequeue (I) | |

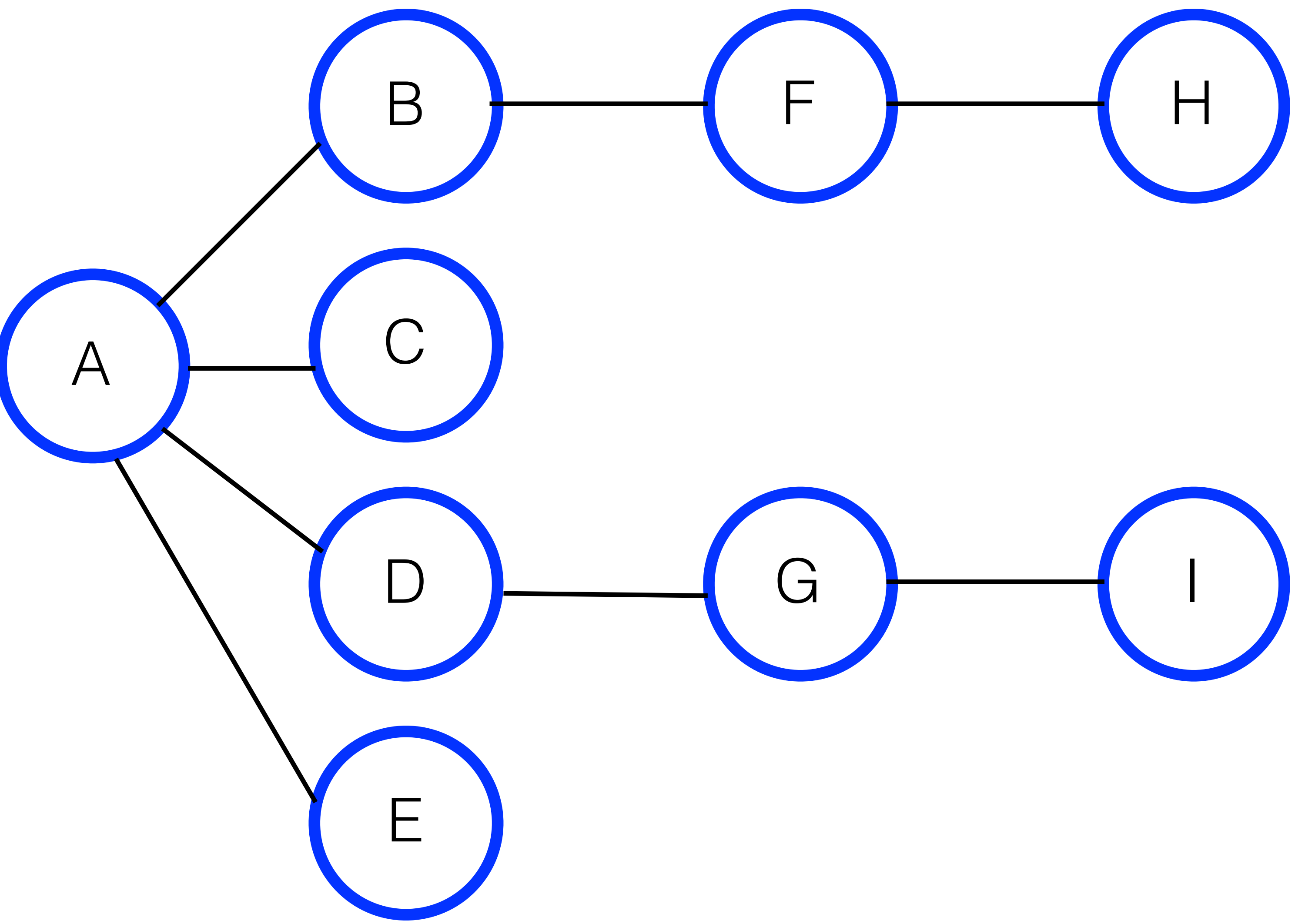



| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |
| Dequeue (C) | DEF |
| Dequeue (D) | EF |
| Visit G | EFG |
| Dequeue (E) | FG |
| Dequeue (F) | G |
| Visit H | GH |
| Dequeue (G) | H |
| Visit I | HI |
| Dequeue (H) | I |
| Dequeue (I) | |

Now the queue is empty, so it is time for **Rule 3:**
"If you can't carry out Rule 2 because the queue is empty, you are finished"




| Event | Queue |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Dequeue (B) | CDE |
| Visit F | CDEF |
| Dequeue (C) | DEF |
| Dequeue (D) | EF |
| Visit G | EFG |
| Dequeue (E) | FG |
| Dequeue (F) | G |
| Visit H | GH |
| Dequeue (G) | H |
| Visit I | HI |
| Dequeue (H) | I |
| Dequeue (I) | |
| Done | |




**Order:** ABCDEFGHI

**Time:** $O(|V| + |E|)$

| Event | Queue |
|---|---|
| **Visit A** | |
| **Visit B** | B |
| **Visit C** | BC |
| **Visit D** | BCD |
| **Visit E** | BCDE |
| Dequeue (B) | CDE |
| **Visit F** | CDEF |
| Dequeue (C) | DEF |
| Dequeue (D) | EF |
| **Visit G** | EFG |
| Dequeue (E) | FG |
| Dequeue (F) | G |
| **Visit H** | GH |
| Dequeue (G) | H |
| **Visit I** | HI |
| Dequeue (H) | I |
| Dequeue (I) | |
| Done | |

# BFS

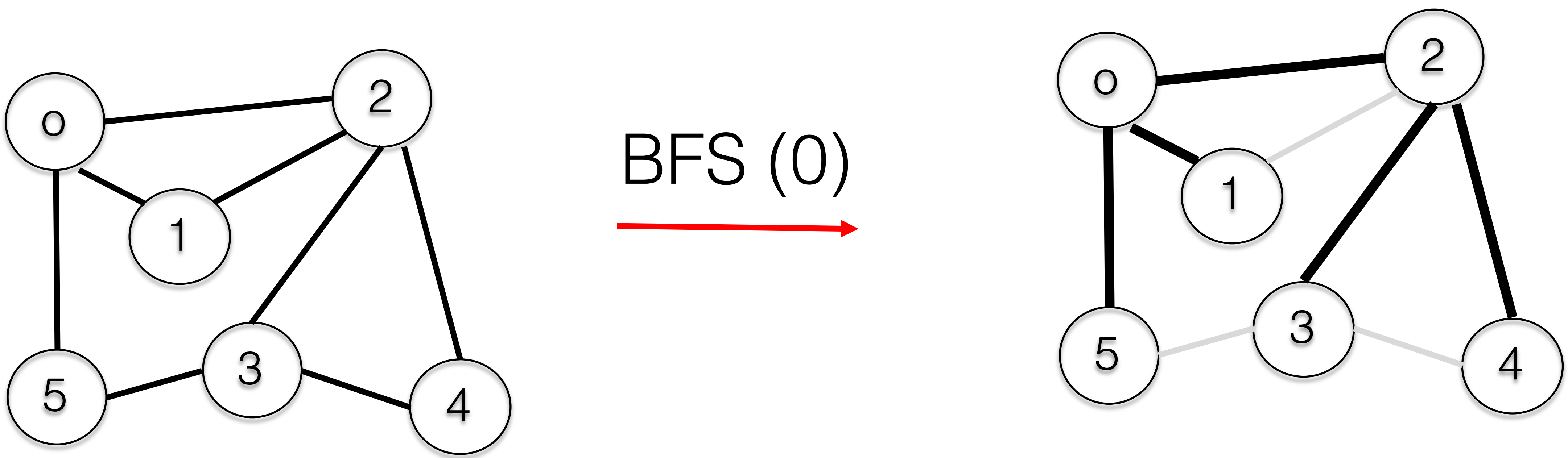
- Notice that,
    - BFS tries to stay as close as possible to the starting point
    - Thus the name, **Breadth First Search**

# BFS

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Cormen, Ch: 22

# Example



# DFS

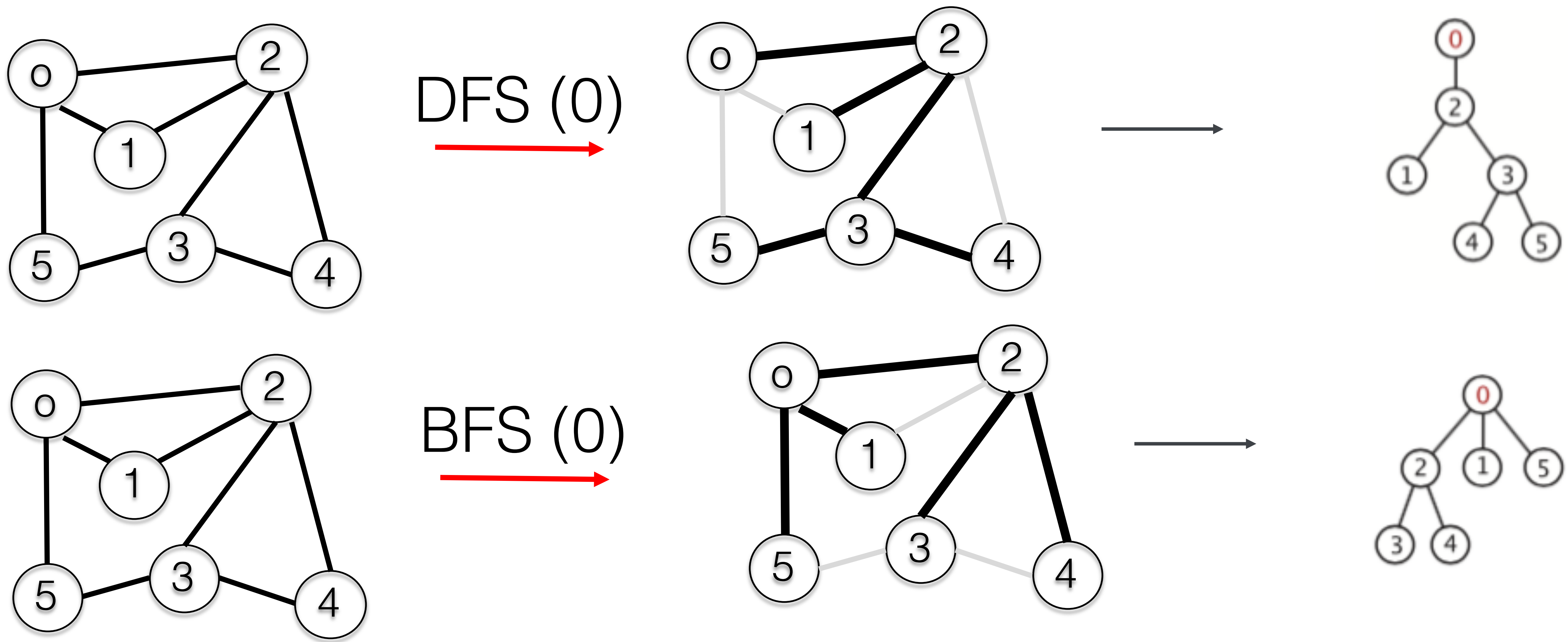
```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1                // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]          // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK                // blacken u; it is finished
time = time + 1
u.f = time
```

Cormen, Ch: 22

# Example

# Final Remarks




DFS finds a path, whereas BFS finds the shortest path

However, note that the graph is: unweighted (or same weight)

# Did we achieve today's objectives?

1. Build a definition for the "connected component of a graph"

2. Learn graph traversals

- Depth First Search (DFS)

- Breadth First Search (BFS)

Introduction to Algorithms, Chapter 22
Data Structures & Algorithms in Java, Chapter 14